Microservice Patterns and Best Practices

微服务设计模式和最佳实践

[美] 维尼休斯·弗多萨·帕切科 著　程晓磊 译



清华大学出版社

微服务设计模式和最佳实践

[美] 维尼休斯 • 弗多萨 • 帕切科　著

程晓磊　译

清华大学出版社

北　京

内 容 简 介

本书详细阐述了与微服务相关的基本解决方案，主要包括微服务概念、微服务工具、内部模式、微服务生态环境、共享数据微服务设计模式、聚合器微服务设计模式、代理微服务设计模式、链式微服务设计模式、分支微服务设计模式、异步消息微服务、微服务间的协同工作、微服务测试以及安全监测和部署方案等内容。此外，本书还提供了相应的示例、代码，以帮助读者进一步理解相关方案的实现过程。

本书适合作为高等院校计算机及相关专业的教材和教学参考书，也可作为相关开发人员的自学教材和参考手册。

北京市版权局著作权合同登记号 图字：01-2018-3298

图书在版编目（CIP）数据

微服务设计模式和最佳实践/（美）维尼休斯・弗多萨・帕切科著；程晓磊译. —北京：清华大学出版社，2019
书名原文：Microservice Patterns and Best Practices
ISBN 978-7-302-52041-2

Ⅰ. ①微… Ⅱ. ①维… ②程… Ⅲ. ①互联网络-网络服务器 Ⅳ. ①TP368.5

中国版本图书馆 CIP 数据核字（2019）第 008130 号

责任编辑： 贾小红
封面设计： 刘 超
版式设计： 魏 远
责任校对： 马子杰
责任印制： 宋 林

出版发行： 清华大学出版社
网 址： http://www.tup.com.cn，http://www.wqbook.com
地 址： 北京清华大学学研大厦 A 座 **邮 编：** 100084
社 总 机： 010-62770175 **邮 购：** 010-62786544
投稿与读者服务： 010-62776969，c-service@tup.tsinghua.edu.cn
质 量 反 馈： 010-62772015，zhiliang@tup.tsinghua.edu.cn
印 装 者： 清华大学印刷厂
经 销： 全国新华书店
开 本： 185mm×230mm **印 张：** 19 **字 数：** 377 千字
版 次： 2019 年 3 月第 1 版 **印 次：** 2019 年 3 月第 1 次印刷
定 价： 99.00 元

产品编号：080091-01

译 者 序

“微服务”的概念是由 ThoughtWorks 公司的首席科学家 Martin 提出的，他是敏捷开发方法创始人之一。自然而然地，微服务的目的就是有效拆分应用，实现敏捷开发和部署。与之相对应的，此前的软件开发都是基于一体化架构。一体化架构的优点是开发简单直接、集中式管理、功能都在本地；但是，一旦应用程序变大、团队扩张，那么一体化架构的缺点就会立即凸显，庞大的一体代码库可能会让新手程序员望而生畏，再加上开发效率低、代码维护困难、部署不灵活、难以扩展应用等，这些很可能成为企业和开发人员难以逾越的障碍。微服务的出现解决了这些矛盾，它将系统拆分成进程独立的服务，进行分布式管理、自动化运维，通过 API 网关、服务间调用、服务发现、服务容错、服务部署和数据调用实现了开发简单、独立按需扩展、高可用性、持续集成和持续交付等，特别契合云平台应用程序开发的需要，这也正是它日益流行的原因。

本书将介绍不同阶段的微服务中应用程序开发的各种设计模式及其最佳实践。微服务模式和最佳实践始于对微服务关键概念的理解，并展示如何在设计微服务时做出正确的选择。本书将讨论内部微服务应用程序的各种方法，如缓存策略、异步机制、CQRS 和事件源等。随着过程的不断推进，读者将深入了解微服务的相关设计模式。

在本书的翻译过程中，除程晓磊外，刘璋、张博、刘晓雪、王烈征、张华臻、刘祎等人也参与了部分翻译工作，在此一并表示感谢。

由于译者水平有限，难免有疏漏和不妥之处，恳请广大读者批评指正。

译 者

前　　言

微服务是一种软件体系结构策略，多年来一直在使用，其目标是提高服务的可伸缩性。由于当今的业务呈现快速、动态增长之势，单体应用程序与面向服务相比已不占优势。通过设计这种新的体系结构模型，面向对象的原则、标准、解耦和职责已经构成了超越自动化测试的基础内容。

微服务可使读者能够创建可维护、可伸缩的应用程序。在阅读本书后，读者将能够创建可互操作的微服务，同时兼具可测试性和高性能的特征。

适用读者

本书面向于具有 Web 开发经验，但想要改进其开发技术以创建可维护和可伸缩应用程序的读者，读者不需要具备微服务方面的经验。本书中演示的概念和标准，使读者能够开发易于理解和支持大量访问的应用程序。

本书内容

第 1 章整体介绍微服务的概念，以帮助读者理解体系结构背后的相关理念，例如客户优先方案以及域定义。

第 2 章讨论微服务应用中的常见工具。一旦了解了客户以及如何定义应用程序域，即可制定技术决策方案，包括所使用的语言、框架，以及如何验证微服务框架的功能。

第 3 章将探讨内部微服务的应用模式，如缓存策略、worker、队列以及异步机制。

第 4 章考察如何从单体应用程序中创建一组具有弹性和可伸缩的微服务。本章还将重点讨论如何在各自的容器中正确地分离微服务，并解释存储层的分布。

第 5 章涉及一些较为特殊的用例。通常，微服务在业务、测试、通信、连接和存储方面是完全独立的；而共享模式则是迁移概念的一种特殊模式（从单体到微服务）；总的来说，这是一种过渡模式。

第 6 章主要讨论了一些简单而常见的模式，主要包括对微服务的数据编排。

第 7 章阐述了代理模式，它与聚合器模式非常相似。对于该结构，当不需要连接数据并将其发送给终端用户时，可以对其加以使用。本章的目标是了解代理模式和聚合器模式之间的差异，以及针对微服务的正确的请求重定向操作。

第 8 章将学习链式模式。其中，发送至客户端的信息将整合至链中。本章的主要目标是解释信息整合问题。

第 9 章介绍分支模式，该模式可视作聚合器模式和链式模式的混合体，通常用于以下场合：在后端中，微服务未包含所有数据以完成某项任务；或者应直接通知另一个微服务。本章解释了该模式的应用时机，以及对业务的作用方式。

第 10 章解释了一种较为复杂的模式：在微服务级别使用异步机制。本章将讨论如何利用消息工具实现微服务的异步通信方式。

第 11 章对相关模式进行总结。在介绍了所有的微服务模式后，将考察如何将微服务进行整合，以使其可有效地协同工作。

第 12 章将讨论最为合理的测试机制及其简化方式。

第 13 章介绍了生产过程中维护微服务的必要条件以及最佳实践。

背景知识

如果读者了解一些 Go 语言 （golang）中的 OOP 和包结构，那么，阅读本书将变得更加轻松、有趣。

资源下载

读者可访问 http://www.packtpub.com 并通过个人账户下载示例代码文件。另外，http://www.packtpub.com/support，注册成功后，我们将以电子邮件的方式将相关文件发与读者。

读者可根据下列步骤下载代码文件。

（1）登录 www.packtpub.com 并注册我们的网站。

（2）选择 SUPPORT 选项卡。

（3）单击 Code Downloads & Errata。

（4）在 Search 文本框中输入书名并执行后续命令。

当文件下载完毕后，确保使用下列最新版本软件解压文件夹。

- ❑ Windows 系统下的 WinRAR/7-Zip。
- ❑ Mac 系统下的 Zipeg/iZip/UnRarX。
- ❑ Linux 系统下的 7-Zip/PeaZip。

另外，读者还可访问 GitHub 获取本书的代码包，对应网址为 https://github.com/PacktPublishing/Microservice-Patterns-and-Best-Practices。此外，读者还可访问 https://github.com/PacktPublishing/以了解丰富的代码和视频资源。

读者可访问 https://www.packtpub.com/sites/default/files/downloads/MicroservicePatternsandBestPractices_ColorImages.pdf 下载本书的彩色图像，以方便读者对比某些输出结果。

本书约定

代码块则通过下列方式设置：

```
class TestDevelopmentConfig(TestCase):

    def create_app(self):
        app.config.from_object('config.DevelopmentConfig')
        return app

    def test_app_is_development(self):
        self.assertTrue(app.config['DEBUG'] is True)
```

代码中的重点内容则采用黑体表示：

```
@patch('views.rpc_command')
def test_sucess(self, rpc_command_mock):
    """Test to insert a News."""
```

命令行输入或输出如下所示：

```
$ docker-compose -f docker-compose.yml up --build -d
```

图标表示较为重要的说明事项。

图标则表示提示信息和操作技巧。

读者反馈和客户支持

欢迎读者对本书的建议或意见予以反馈。

对此，读者可向 feedback@packtpub.com 发送邮件，并以书名作为邮件标题。若读者对本书有任何疑问，均可发送邮件至 questions@packtpub.com，我们将竭诚为您服务。

勘误表

尽管我们在最大程度上做到尽善尽美，但错误依然在所难免。如果读者发现谬误之处，无论是文字错误抑或是代码错误，还望不吝赐教。对此，读者可访问 http://www.packtpub.com/submit-errata，选取对应书籍，单击 Errata Submission Form 超链接，并输入相关问题的详细内容。

版权须知

一直以来，互联网上的版权问题从未间断，Packt 出版社对此类问题异常重视。若读者在互联网上发现本书任意形式的副本，请告知网络地址或网站名称，我们将对此予以处理。关于盗版问题，读者可发送邮件至 copyright@packtpub.com。

若读者针对某项技术具有专家级的见解，抑或计划撰写书籍或完善某部著作的出版工作，则可访问 www.packtpub.com/authors。

问题解答

若读者对本书有任何疑问，均可发送邮件至 questions@packtpub.com，我们将竭诚为您服务。

目　　录

第 1 章　微服务概念 1
1.1　理解应用程序 2
1.1.1　领域驱动设计 2
1.1.2　单一职责原则 4
1.1.3　显式发布的接口 5
1.2　独立部署、更新、扩展以及替换 7
1.2.1　独立部署 7
1.2.2　更新 7
1.2.3　可扩展性 8
1.3　轻量级通信 12
1.3.1　同步 13
1.3.2　异步 13
1.4　异质/多语言 14
1.5　通信的文档化 14
1.6　Web 应用程序端点 15
1.7　移动应用程序端点 15
1.8　缓存客户端 16
1.9　调节客户端 17
1.10　确定贫血域 17
1.11　确定 fat 域 18
1.12　针对业务确定微服务域 18
1.13　从域到实体 19
1.14　本章小结 20
第 2 章　微服务工具 21
2.1　编程语言 21
2.1.1　熟练程度 22
2.1.2　性能 22
2.1.3　实践开发 23

2.1.4　生态圈 23
2.1.5　扩展性的开销 24
2.1.6　选取编程语言 24
2.2　微服务框架 27
2.2.1　Python 语言 27
2.2.2　Go 语言 29
2.3　二进制通信——服务间的直接通信 31
2.3.1　理解通信方式 31
2.3.2　直接通信间的警示信息 35
2.4　消息代理——服务间的异步通信 37
2.4.1　ActiveMQ 38
2.4.2　RabbitMQ 39
2.4.3　Kafka 40
2.5　缓存工具 40
2.5.1　Memcached 42
2.5.2　Redis 42
2.6　故障警示工具 44
2.6.1　性能 44
2.6.2　构建 45
2.6.3　组件 46
2.6.4　实现鸿沟 47
2.7　数据库 47
2.8　本地性能度量 48
2.8.1　Apache Benchmark 49
2.8.2　WRK 50
2.8.3　Locust 51
2.9　本章小结 53
第 3 章　内部模式 55
3.1　开发结构 55
3.1.1　数据库 55
3.1.2　编程语言和工具 56
3.1.3　项目结构 56
3.2　缓存策略 71

3.2.1 缓存机制的应用 72
3.2.2 缓存优先 78
3.2.3 队列任务 79
3.2.4 异步机制和 worker 81
3.3 CQRS——查询策略 87
3.3.1 CQRS 的概念 87
3.3.2 理解 CQRS 88
3.3.3 CQRS 的优点和缺陷 90
3.4 事件源——数据完整性 91
3.5 本章小结 92
第 4 章 微服务生态环境 93
4.1 容器中的分离机制 93
4.1.1 分层服务架构 95
4.1.2 分离 UsersService 96
4.2 存储分布 103
4.2.1 折旧数据 103
4.2.2 区域化数据 103
4.3 隔离——使用生态系统防止故障的出现 104
4.3.1 冗余设计 104
4.3.2 临界分区 109
4.3.3 隔离设计 110
4.3.4 快速故障 111
4.4 断路器 112
4.5 本章小结 113
第 5 章 共享数据微服务设计模式 115
5.1 理解模式 115
5.2 将单体应用程序划分为微服务 116
5.2.1 定义优先级 117
5.2.2 设置期限 117
5.2.3 定义应用程序域 117
5.2.4 试验操作 117
5.2.5 制定标准 118

5.2.6　构建原型 118
5.2.7　发送产品 118
5.2.8　开发新的微服务 118
5.3　数据编排 130
5.4　响应整合 132
5.5　微服务通信 132
5.6　存储共享反模式 133
5.7　最佳实践 133
5.8　测试机制 133
5.9　共享数据模式的利弊 135
5.10　本章小结 136
第 6 章　聚合器微服务设计模式 137
6.1　理解聚合器设计模式 137
6.2　使用 CQRS 和事件源 139
6.2.1　分离数据库 139
6.2.2　重构微服务 140
6.3　微服务通信 153
6.3.1　创建编排器 154
6.3.2　使用消息代理 159
6.4　模式扩展 163
6.5　瓶颈反模式 164
6.6　最佳实践 166
6.7　测试 167
6.7.1　功能测试 167
6.7.2　集成测试 168
6.8　聚合器设计模式的优缺点 170
6.8.1　聚合器设计模式的优点 170
6.8.2　聚合器设计模式的缺点 170
6.9　本章小结 170
第 7 章　代理微服务设计模式 171
7.1　代理方案 171
7.1.1　哑代理 172

7.1.2　智能代理 172
7.1.3　理解当前代理 173
7.2　编排器的代理策略 175
7.3　微服务通信 176
7.4　模式扩展性 176
7.5　最佳实践 177
7.5.1　纯粹的模式 177
7.5.2　瓶颈问题 178
7.5.3　代理制的缓存机制 178
7.5.4　简单的响应 178
7.6　代理设计模式的优缺点 179
7.7　本章小结 179

第 8 章　链式微服务设计模式 181

8.1　理解模式 181
8.2　数据编排和响应整合 184
8.3　微服务通信 185
8.4　模式扩展性 185
8.5　“大泥球”反模式 186
8.6　最佳实践方案 188
8.6.1　纯微服务 188
8.6.2　请求一致性数据 188
8.6.3　深入理解链式设计模式 189
8.6.4　关注通信层 189
8.7　链式设计模式的优缺点 189
8.8　本章小结 190

第 9 章　分支微服务设计模式 191

9.1　理解模式 191
9.2　数据编排和响应整合 194
9.3　微服务通信 195
9.4　模式扩展 197
9.5　最佳实践方案 198
9.5.1　域定义 198

9.5.2 遵守规则 198
9.5.3 关注物理组件 198
9.5.4 简化行为 199
9.6 分支设计模式的优缺点 199
9.7 本章小结 199
第 10 章 异步消息微服务 201
10.1 理解当前模式 201
10.2 域定义——RecommendationService 203
10.3 域定义——RecommendationService 204
10.4 微服务编码 204
10.5 微服务通信 211
10.5.1 使用消息代理和队列 211
10.5.2 准备 pub/sub 结构 212
10.6 模式的可扩展性 214
10.7 进程序列反模式 214
10.8 最佳实践方案 215
10.8.1 应用程序定义 215
10.8.2 不要尝试创建响应 216
10.8.3 保持简单性 216
10.9 异步消息传递设计模式的优缺点 216
10.10 本章小结 217
第 11 章 微服务间的协同工作 219
11.1 理解当前应用程序状态 219
11.1.1 公共饰面层 220
11.1.2 内部层 222
11.1.3 理解通用工具 223
11.2 通信层和服务间的委托 224
11.2.1 理解服务间的数据合约 225
11.2.2 使用二进制通信 228
11.3 模式分布 235
11.4 故障策略 236
11.5 API 集成 237

11.6 本章小结 239

第 12 章 微服务测试 241

12.1 单元测试 241
12.2 针对集成测试配置容器 249
12.3 集成测试 251
12.4 端到端测试 253
12.5 发布管线 259
12.6 签名测试 259
12.7 Monkey 测试 260
12.8 Chaos Monkey 260
12.9 本章小结 262

第 13 章 安全监测和部署方案 263

13.1 监测微服务 263
13.1.1 监测单一服务 264
13.1.2 监测多项服务 266
13.1.3 查看日志 267
13.1.4 应用程序中的错误 268
13.1.5 度量方法 271
13.2 安全问题 272
13.2.1 理解 JWT 272
13.2.2 单点登录 275
13.2.3 数据安全 276
13.2.4 预防恶意攻击——识别攻击行为 277
13.2.5 拦截器 277
13.2.6 容器 278
13.2.7 API 网关 279
13.3 部署 279
13.3.1 持续集成和持续交付/持续部署 280
13.3.2 蓝/绿部署模式和 Canary 发布 281
13.3.3 每台主机包含多个服务实例 282
13.3.4 每台主机的服务实例 283
13.4 本章小结 285

第 1 章　微服务概念

在编程领域，设计模式十分常见。同样，在 Web 开发中，这一点也不例外。随着互联网的普及以及 Web 2.0 的出现，许多为 Web 开发的模式被广泛传播，目的是使开发更动态、更简单，以适应新的特性。

诸如 MVC（模型-视图-控制器）、HMVC（层次模型视图控制器）和 MTV（模型模板视图）等模式激发了各种框架的创建，例如 Django、Ruby on Rails、Spring MVC 和 CodeIgniter。所有这些框架都非常适合快速创建 Web 应用程序，且无须对应用程序架构予以更多的关注。这是因为大部分工作都是由框架完成的，所有这些模式适用于包含所需业务规则的 Web 应用程序。通常，这些应用程序（其中，所有业务规则都位于同一代码库上）被称为单体。多年以来，在 Web 开发生态系统中，单体绝对占统治地位。许多公司通过全栈框架软件产品在市场上寻找销售空间。许多单体软件已大量应用于互联网上，随着时间的推移，这些单体应用也产生了一些问题。

对于单体应用程序来说，其开发过程存在以下困难：

- 鉴于整合操作难以实施，同一代码库中的维护变得更加复杂。
- 实现新特性的难度不断增加，开发周期往往超出预期的规定。
- 应用程序的可伸缩性。
- 部署新特性且不会对在线内容产生影响变得越加困难。
- 体系结构的变化使问题趋于复杂化。

上述内容只是单体应用程序中可能存在的部分问题。这些困难也使得人们思考，并将应用程序架构迁移到微服务中去。越来越多地，微服务已经成为软件工程行业和大多数公司所采取的方案。在这些行业中，“成功”一词意味着实践过程和可伸缩业务所面临的问题。微服务体系结构包含以下优点：

- 针对每个微服务设置专有的业务领域，以促进新特性的实现。
- 更好的业务定义，消除了它们之间的循环依赖关系。
- 独立部署。
- 简化错误的识别过程。
- 微服务之间技术的独立性。
- 团队间的独立性。
- 实现隔离。

- 针对特定微服务的可伸缩性。

本书旨在向读者探讨如何从单体应用程序过渡到微服务，应用适当的模式并向读者展示微服务的实现过程。相关内容涉及具有交互性的新闻门户网站、推荐系统、身份验证和授权系统。在本书中，当应用设计模式时，读者将经历迁移过程中的每一个步骤；当涉及微服务之间的通信层时，还会了解到内部和外部迁移过程中的具体实施方案。对此，读者首先需要了解一些重要的概念，以便有效地实现微服务。

1.1　理解应用程序

本书中的应用程序将成为所有概念解释和实际代码的来源。具体来说，基本系统是一个新闻门户网站，主要包括以下 3 方面内容：

- 新闻。
- 推荐系统。负责存储用户偏好设置，因此能够向用户提供特定的新闻，甚至根据用户配置文件形成唯一的主页。
- 用户，即基本的注册用户信息。

应用程序的所有业务内容都位于相同的源代码上，即单体软件。该应用程序是在 Django 框架上开发的，使用 PostgreSQL 作为数据库，并使用 Memcached 作为缓存系统（仅应用于数据库层）。

在此基础上，如果推荐级别超出负荷，那么所有应用程序都必须调整其规模，而不仅仅是引用推荐的部分，因为当前应用程序采用了单体结构。对应用程序来说，栈中所发生的变化其代价相对高昂。如果用户想更改所使用的缓存类型，那么所有的其他缓存内容都将丢失。

1.1.1　领域驱动设计

OOP 的概念也适用于微服务设计，这不仅体现于应用程序设计，还包括体系结构和业务逻辑的划分，但对于领域驱动设计（DDD）来说，情况则变得简单起来。

DDD 这一概念源自 Eric Evans 撰写的 Domain-Driven Design 一书。书中内容源自作者二十多年来 OOP 软件开发经验，其中列举了大量的模式分类。需要注意的是，OOP 不仅存在于继承机制、接口中，同时还涉及其他方面，如下所示：

- 代码与业务的一致性。
- 复用性。

- ❑ 最小耦合。

Domain-Driven Design 一书分为以下 4 个部分：

（1）使用域模型。

（2）模型构造块驱动设计。

（3）重构以深入理解模型。

（4）策略设计。

上述各项内容均适用于微服务。其中，（2）、（3）项可用于代码自身；其他项可以在软件内部中应用，也可以用于设计微服务及其签名。

第一部分强调了业务负责人和开发团队之间进行语言交流的必要性。这种通用语言包含了业务专家和开发团队之间日常会话的部分内容。每个人都应该在言语、源代码和微服务签名中使用相同的术语。这意味着，当业务专家说，主页应该用标题和描述内容显示突发新闻，相应地，代码中的语言将表示如下：

- ❑ 微服务新闻。
- ❑ 端点最近的新闻。
- ❑ 带有属性标题和描述的有效载荷。

这种类型的沟通方式可减少理解需求条件时所出现的错误，同时维护与应用程序相关的通识。另外，在使用通用语言时，领域识别将变得更加简单——交互过程使用了标准化的术语。

上述第（4）项将描述微服务的边界以及双方之间的管理便利性。在开发具有一致性和有效性的微服务时，除了使用通用语言之外，还需要一些策略来处理复杂的系统。其中，多个软件组成部分（由多个团队开发）彼此间进行交互。另外，还需要划分每个团队工作的上下文环境，以及这些团队与环境之间的交互程度。在 DDD 提供给我们的众多工具中，下列 3 种工具针对高效的微服务表现得较为突出。

- ❑ 上下文映射：表示为微服务之间的通信路径，微服务团队之间应存在适当的交互。在定义了领域分析后，当前团队可以选择依赖另一个团队来使用领域语言。
- ❑ 反腐坏层（ACL）：负责将外部概念转换为内部模型，以提供域之间的松散耦合。
- ❑ 交换上下文环境：为两个团队提供了一个环境，讨论每个外部术语的含义，并负责翻译微服务的语言。

针对正在开发的应用程序，以及如何避免微服务工作方式解释时出现的错误，DDD 非常有用。新闻系统中一个常见的误解是关于用户和作者这两个术语：二者都是系统的用户，一个作为参与者，另一个作为发布者。然而，如果产品所有者说，“用户发布了坏消息”，此时，产品团队和开发团队之间的沟通上就有问题了。这可能导致业务自身内的、不一致的微服务。另一个问题是，产品所有者先前发表的言论表达了一个不需要

的特性，即可以发布素材的用户。DDD 仅考察如何定义微服务域和标准化术语，以生成一致的内部模型和具有实际含义的 API。同一属性的含义或表达差异称为语义鸿沟（semantic gap），这也是本章要处理的问题。

除了前面提到的内容可帮助创建完整的微服务之外，在设计微服务时，还涉及一个较为重要的特性。边界上下文的概念对于确定微服务的范围，以及最终微服务所具有的职责是必不可少的。读者需要着重理解的是，如果缺少此类条件，耦合限制将会很高，并且像单一职责这样的概念将永远无法实现。

1.1.2　单一职责原则

另一个适用于微服务的原则来自于 OOP 领域，具体来说就是 SOLID 中的字母 S。在类级别之前应当思考的则是应用程序级别。微服务在域级别上是非常微小的。

微服务域不可太大。相反，它必须受到一定限制。DDD 中的限制条件在使用时其强度将有所提升。准确地讲，微服务域中的限制条件将使得应用程序对变化更加敏感，并对错误的感知更加直接。

在域中维护微服务通常较为困难。无论是出于习惯还是被忽略，人们的自然倾向是试图将所有的业务规则或者相似的代码归类于同一微服务中，甚至不知道它们是否处于相同的域。

为了进一步阐明上述观点，下面考察我们所处理的应用程序，即新闻门户网站。其中，新闻和推荐系统处于协同工作状态。推荐系统负责整合具有相关标记的某些新闻。首先，推荐系统总是与新闻相关联，显然，这不会产生任何问题。同时还会减少出现问题的概率，例如网络延迟。对应的主要概念如图 1.1 所示。

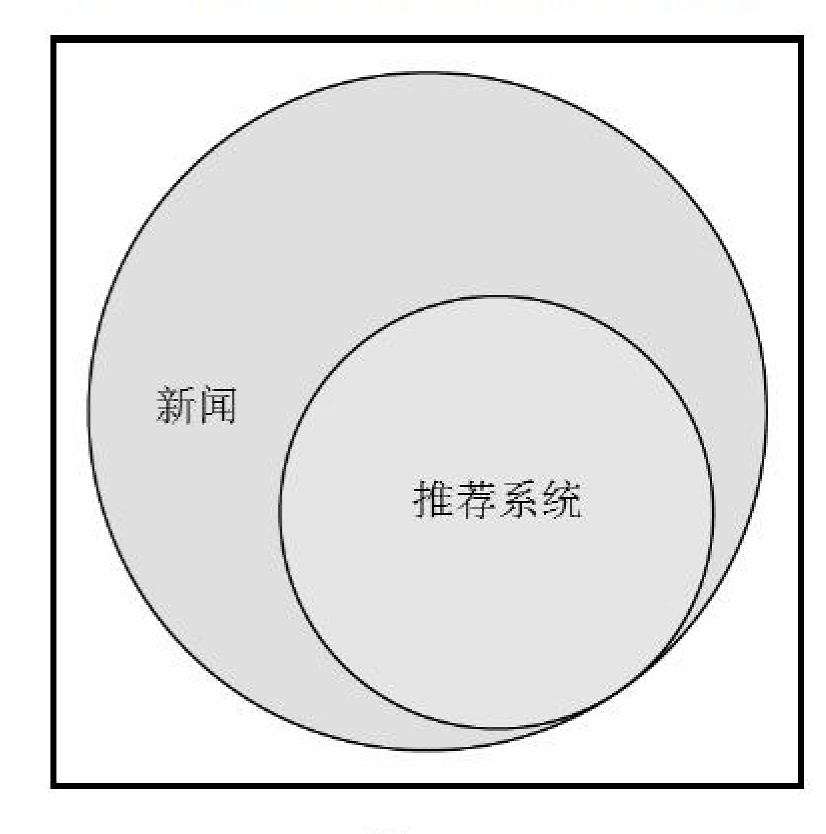


图 1.1

然而，针对未来变化，创建包含新闻和推荐系统的微服务往往会涉及不必要且代价高昂的开销。在简单的演示示例中，我们将查看一些业务变化内容，及其所带来的问题。

此时，新的业务需求出现了。产品负责人可能会思考，是否可根据门户网站上最引人注目的新闻，使用推荐系统来撰写个人主页。因此，推荐系统将不再仅仅与新闻连接，同时还包括用户。根据这一新的需求，与耦合相关的问题将会在部署、扩展以及维护功能代码时浮出水面。

根据上述最新提出的需求条件，不难发现，新闻和推荐系统彼此协同工作，但彼此完全处于不同的域。对于应用程序体系结构来讲，针对各项微服务使用单一职责原则辨识域则十分重要。图 1.2 显示了微服务分布的主要概念。

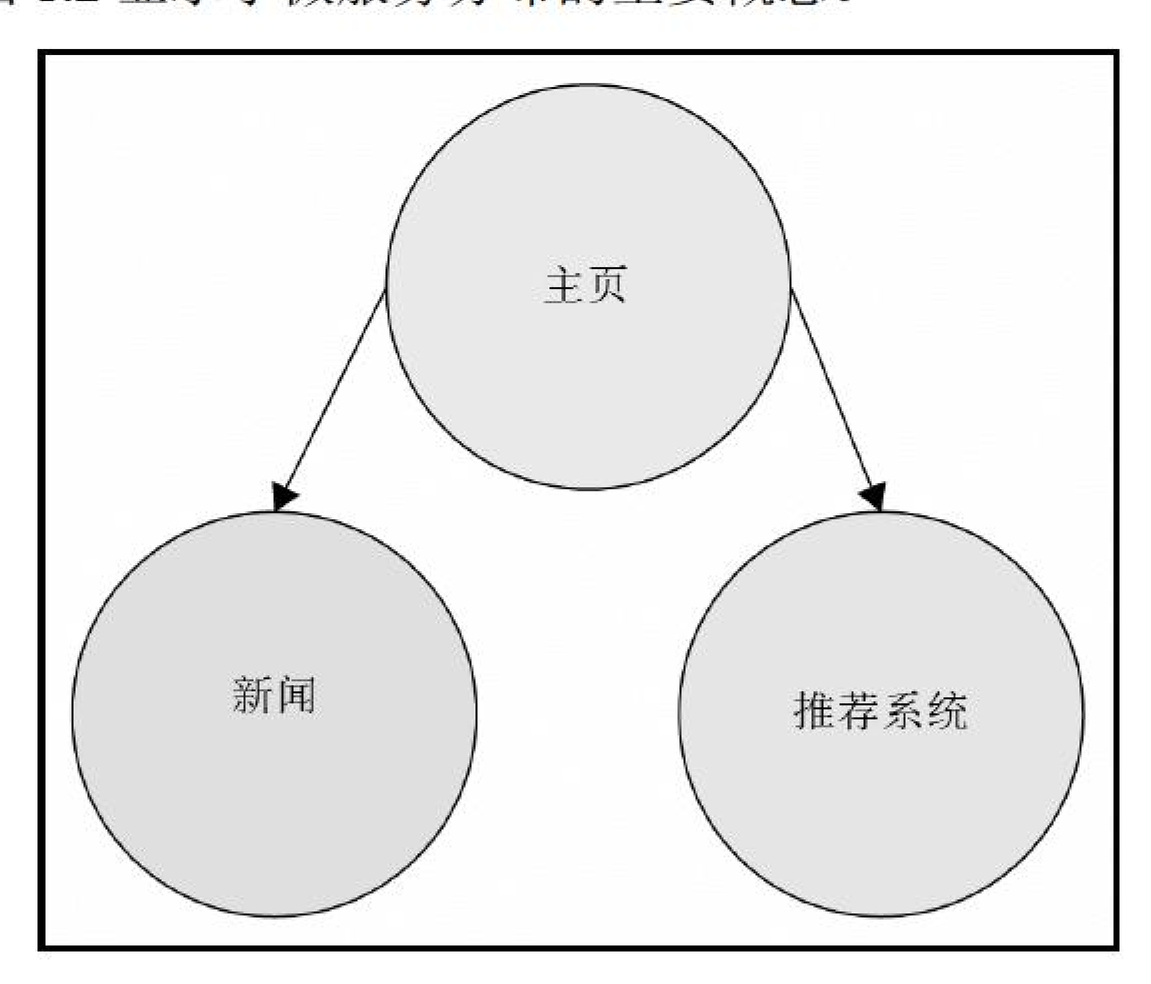


图 1.2

当采取正确方式协同工作时，主页请求源自新闻或推荐系统的信息。本书稍后将介绍一些数据整合形式，但是现在很重要的一点是，读者应理解主页可以请求独立的服务，并方便地对所接收到的数据进行整合。

新闻和推荐系统的分离使得应用程序的部署变得更简单，并生成更一致的代码库，以及为某项功能定义完备和特定的业务域。

1.1.3　显式发布的接口

发布的接口是一个术语，通常会与公共接口产生混淆。理解这两个术语之间的差异至关重要，即微服务和分布式源系统。

对于微服务，全部内部微服务代码在开发团队中共享使用；类方法均定义为抽象类型

或属性，可以是团队之间公共接口的一部分内容，进而可方便地在重构事件中通知、生成变化内容。在开发级别上设定大量的层级结构并不可取，一切均为了在实现中获取速度。

而发布的接口则有所不同。发布的接口由微服务开发人员所发布并被互联网所使用。单点登录（SSO）API 就是一个很好的例子。想象一下，API 为了实现新的特性（如安全性）突然发生变化，并且这些变化并没有为 API 用户提供良好的提示系统。因此，此处使用 SSO 服务并不适宜——鉴于更新行为，API 客户端将面临不兼容问题。

发布的接口应具备更多的控制权，且对重构行为表现得更富有弹性。通常情况下，它们仅应用于外部应用程序客户端上。其间，签名中的变化越少，则效果也就越好。图 1.3 显示了发布的接口签名的维护状态。

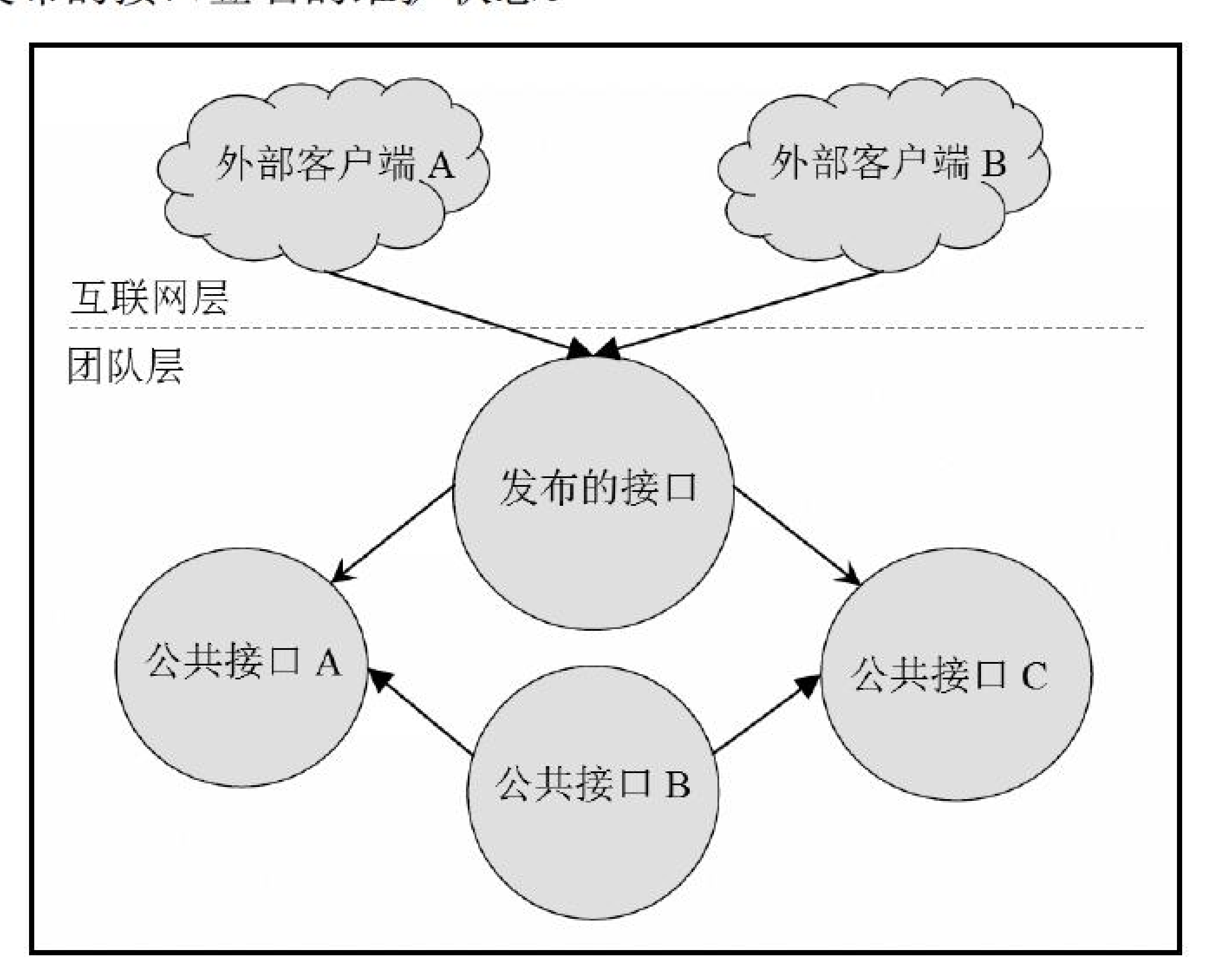


图 1.3

对于发布的接口来说，一些概念需要引起足够的重视，其中包括：

- 发布版本化的接口。高效的版本控制十分关键，并借此指明所弃用的版本。不仅于此，版本控制还用于指定新版本，以及弃用版本何时将被永久停用。
- 小型发布接口。与专用载荷相比，较大的载荷更容易受到变化的影响。在此类载荷上使用 DDD 这一概念则十分恰当。
- 发布的外部接口。不要针对内部开发团队制定发布接口这一概念，这将减缓修改和实现过程。

人们往往将公共接口和发布的接口之间的关系视为与公有和私有 OOP 类似，但是它们实际上是存在差异的。发布的接口并不意味着剥夺客户端资源，而是指示客户端灵活

地使用足够的资源。

1.2　独立部署、更新、扩展以及替换

这一话题较为有趣，且与微服务生态圈有关。与单体系统相比，部署、更新、替换以及扩展体现了微服务的优势。

1.2.1　独立部署

版本控制是专业软件开发中的标准内容，这是因为开发人员几乎在同一应用程序中同时处理遗留代码的特性和维护工作。

最后，可在应用程序中创建一个标签，并将该标签发送到产品中，这一过程称为部署。与此同时，也可能会出现某些问题。

下面考察我们的新闻门户网站。其中，一名开发人员处理推荐系统的某个重要功能，而另一名则负责修复 bug。双方均承诺达到同一目标。在部署时，若新闻中的 bug 并没有被成功修复，这将使得新特性无法置入产品中。软件可以被完整地测试，即使对每项任务给予了很大的关注，不可预见的事情仍然会发生。

对于微服务而言，此类问题将会大大缓解。下面重新考察上述方案。

在新闻门户网站中，一名开发人员针对推荐系统微服务处理重要的功能，而另一名开发人员则处理新闻微服务中的 bug 问题。二者在各自的微服务中完成相同的目标。假设仍在部署过程中遭遇了某个问题，即新闻中的 bug 并未被成功修复，进而无法将新闻微服务的最新版本置于产品中。但推荐系统微服务则不会受到任何影响，并可被正常地置入产品中。

这可能是独立部署的主要优点之一。当然，维护多个机器实例操作的复杂性将会使问题变得更加复杂。但在云计算世界中，即使应用程序是单例程序，多实例的复杂性也不会有太大的变化，因为扩展需求确实存在。

本书后续章节还将介绍部署的模式，并重点讨论如何降低复杂度以及实用性问题，以持续地执行部署任务。

1.2.2　更新

微服务的一项强制性任务则是独立更新问题，其间须遵循相关规则，以确保尽可能地实现独立更新，其中包括：

- ❑ 不要在微服务间共享库。这意味着每个微服务都有一个完全独立于任何其他微服务的堆栈。共享库是在部署时产生高耦合和问题的错误根源。微服务可以始于相同堆栈，尽管一种较好的方法是分析域和数据结构，以查看堆栈是否兼容。但是，从同一个堆栈开始并不意味着保持并发版本控制。另一个需要注意的问题是，应完全避免在库堆栈的特定版本上创建业务组件。这种方法阻止了微服务中的任何技术开发行为。例如，无法使用安全补丁。
- ❑ 微服务域的边界。前述内容曾对边界上下文有所提及，但这一问题仍值得再次强调。当确定微服务域与微服务体系结构是否兼容时，以及所设计的内容是否为解耦的单体部分时，微服务限制十分有用。松散耦合定义了一个受业务更新和变化影响的微服务，而不会与插入微服务的生态系统发生重大冲突。
- ❑ 在微服务间建立客户端-服务器关系。这表明，每个微服务表示为一个独立的应用程序，且自身包含完整的自主性。当某个微服务依赖于另一个微服务的业务解决方案时，需要设置一个提示点。微服务可以自由地相互通信以获取信息，但不能解决业务问题。当微服务向另一个微服务发送消息，并等待响应以完成任务时，将会出现错误。该错误非常重要，将导致可伸缩性和事务处理问题。当微服务向另一个微服务发送消息时，异步问题往往不可忽略。期间，一个微服务服务器执行任务并提供信息；另一个客户端微服务则请求信息。当服务器和客户端链接至某个微服务上时，即会产生设计错误。
- ❑ 在单独的容器中部署。这种方法不仅实现了微服务的独立结构，而且确保微服务的故障是完全独立的，从而不会破坏整个微服务池。当讨论独立容器时，不一定是指虚拟化问题。这里所指的容器可以是物理容器，这取决于公司的策略和资源。然而，将多个微服务置于同一容器中并非是一种良好的方案。需要注意的是，单一容器中的一组微服务在出现循环微服务时将会产生故障。

单一容器对于堆栈工具升级也很重要，但是此类工具并没有经过适当的编码，如数据库和缓存。

1.2.3　可扩展性

可扩展性是一类常见问题，相关内容可参考 Martin L. Abbott 和 Michael T. Fisher 撰写的 *The Art of Scalability* 一书。相应地，可扩展立方体完全适用于微服务；一般情况下，Web 应用程序也需要具备可扩展性。图 1.4 显示了可扩展立方体。

可扩展立方体这一概念表明，基本上存在 3 种形式的扩展行为，即 X 轴、Y 轴和 Z 轴。为了更好地理解这 3 种方案，我们将通过相关示意图进行解释。

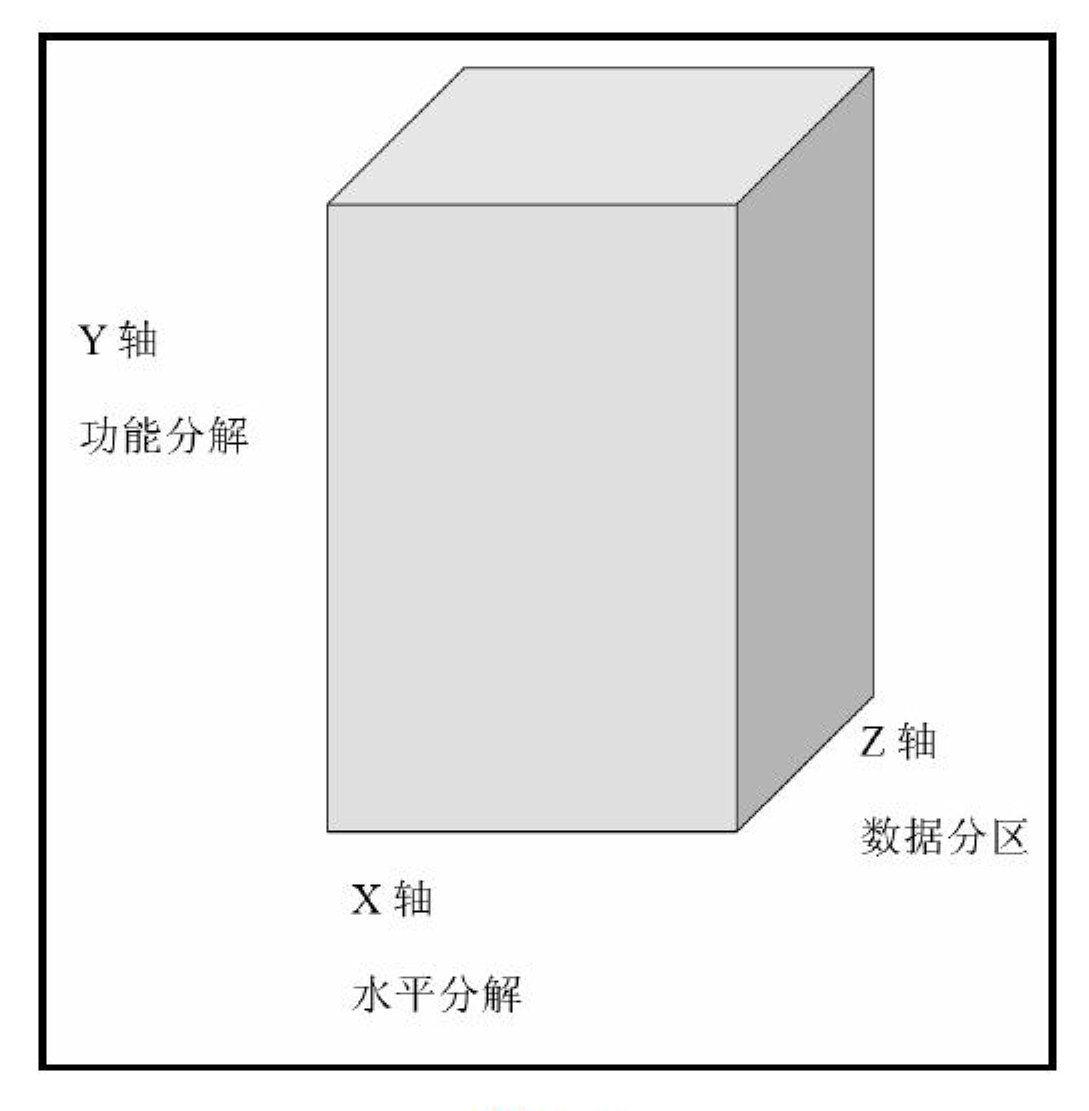


图 1.4

1．X 轴

在 X 轴上，该策略的目标是水平可扩展性，并使用相同的应用服务器（完全复制 *n* 次，并采用 1/*n* 平衡阶）。

该策略的主要问题在于，需要使用到数据库和缓存这一类资源。某些场合下，由于访问这些特性的应用程序的数量会逐渐增加，因此需要进行扩展。针对这种策略，缓存需要更多的内存空间；数据库则需要更大的连接池，因而是否能够带来收益还有待商榷，如图 1.5 所示。

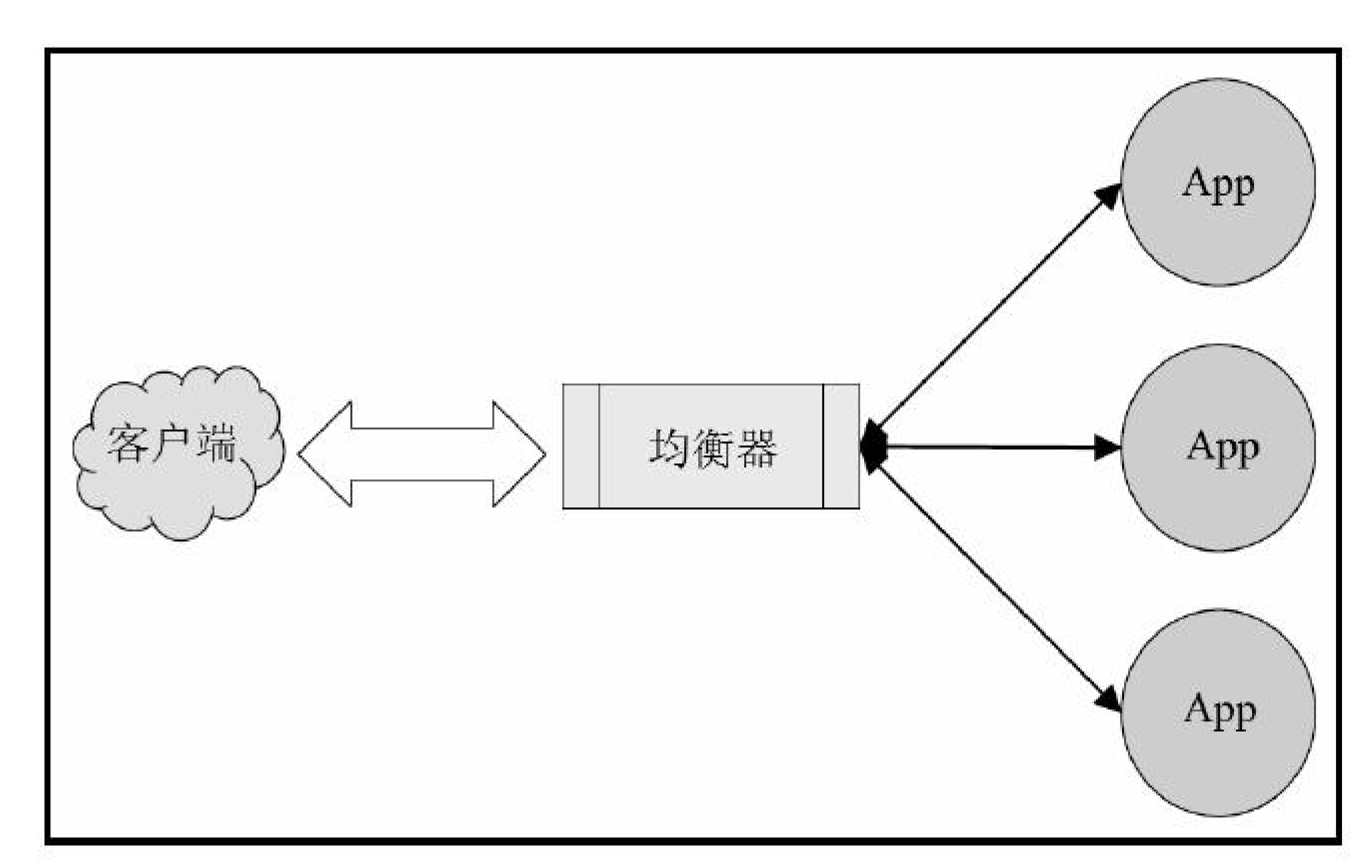


图 1.5

2. Y 轴

在该策略中，均衡器使用一个动词或路线来识别请求的位置。图 1.6 显示了 Y 轴。

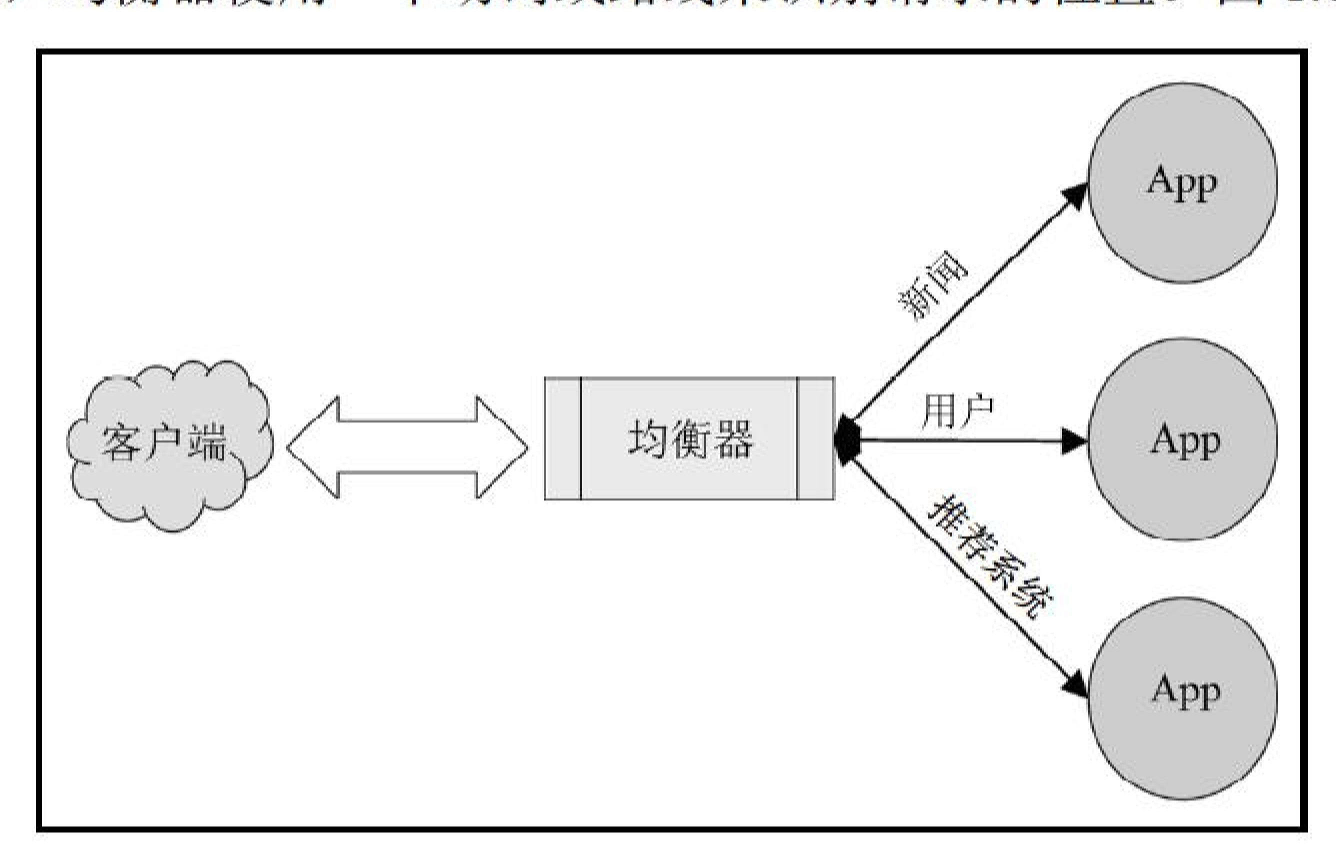


图 1.6

这一原则似乎不太具有可扩展性，但它实际上是 Y 轴和 X 轴的连接点，并可用于扩展微服务。

偶尔情况下，Y 轴和 X 轴之间的这种连接方式可为部分微服务带来可扩展性。在图 1.7 中可以看出，新闻是规模最大的微服务，其次是推荐系统，但用户并没有什么太大的变化。这种扩展技术可极大地消除共享资源访问所带来的问题，其原因在于，每个微服务结构管理并使用自身的资源，例如缓存和数据库。

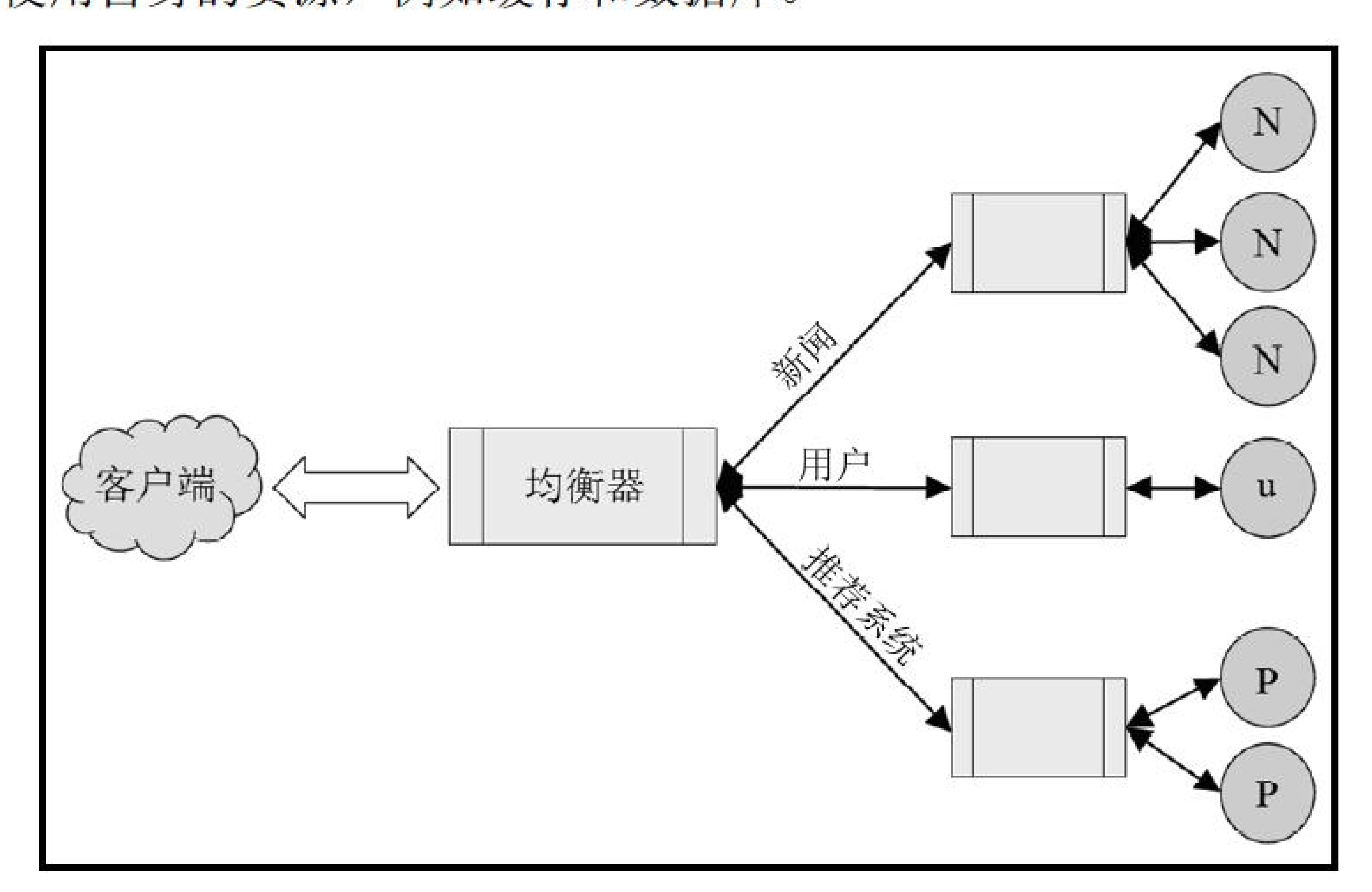


图 1.7

3．Z 轴

对于可扩展结构来说，Z 轴和 X 轴十分类似，因为它在每个服务器上发布的代码完全相同。二者间重要的差别在于，每台服务器响应于特定的数据子集。在该策略中，搜索不仅提供了应用程序的可扩展性，同时还提供了用户所使用的数据。

图 1.8 显示了与 Z 轴相关的示例。

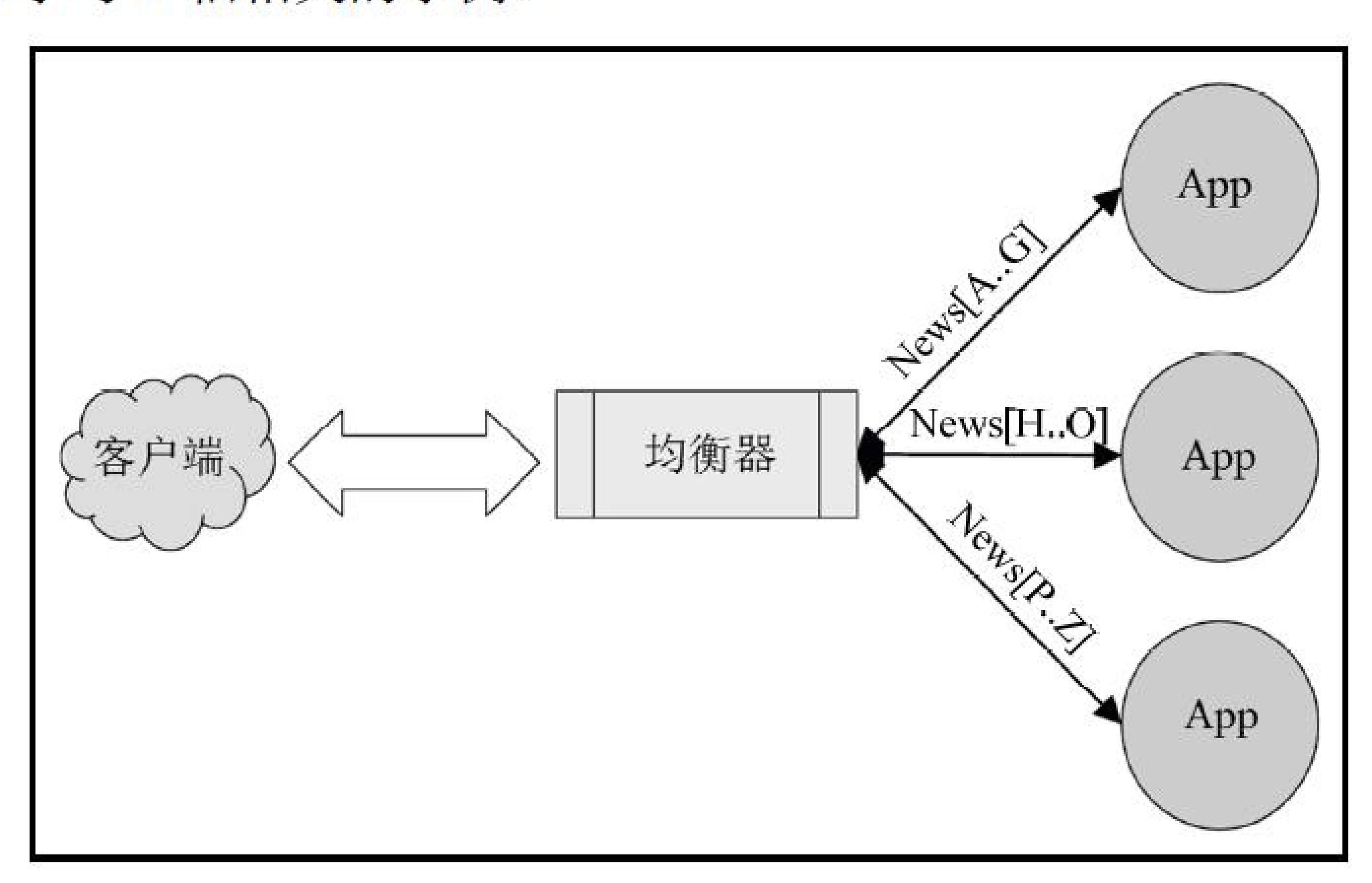


图 1.8

当涉及微服务时，并不完全排除这种策略，但其使用方式则有所不同。适用性最终体现在地理位置上。这意味着，在全球应用程序中，微服务的数据库是按区域分布的，最好是在该区域中可用。也就是说，访问欧洲网站的用户最好能看到欧洲新闻。

如何扩展微服务的定义与业务策略直接相关。从技术的角度来看，重点是提供一种灵活的软件策略，并允许出现变更。

4．替换

微服务更新是一类较为常见的行为，但是有时候这些更新会对微服务产生负面作用。新功能可以使微服务承担许多职责，且超出原有的域概念。

一种常见的错误是添加新特性并使旧功能失效，但并未完全删除它们。当创建一个新的微服务以替换旧的微服务时，开发过程中的某些特性将变得更加清晰。

这一过程可能看起来比较耗时，但是，对于整体应用程序来说，它是非常健康的。另外，还需要重新考虑旧的特性是否仍然有意义，并删除任何与业务无关的僵尸代码，同时成为资源的消费者和复杂度的聚合器。

对于微服务来说，替换过程十分简单，如图 1.9 所示。

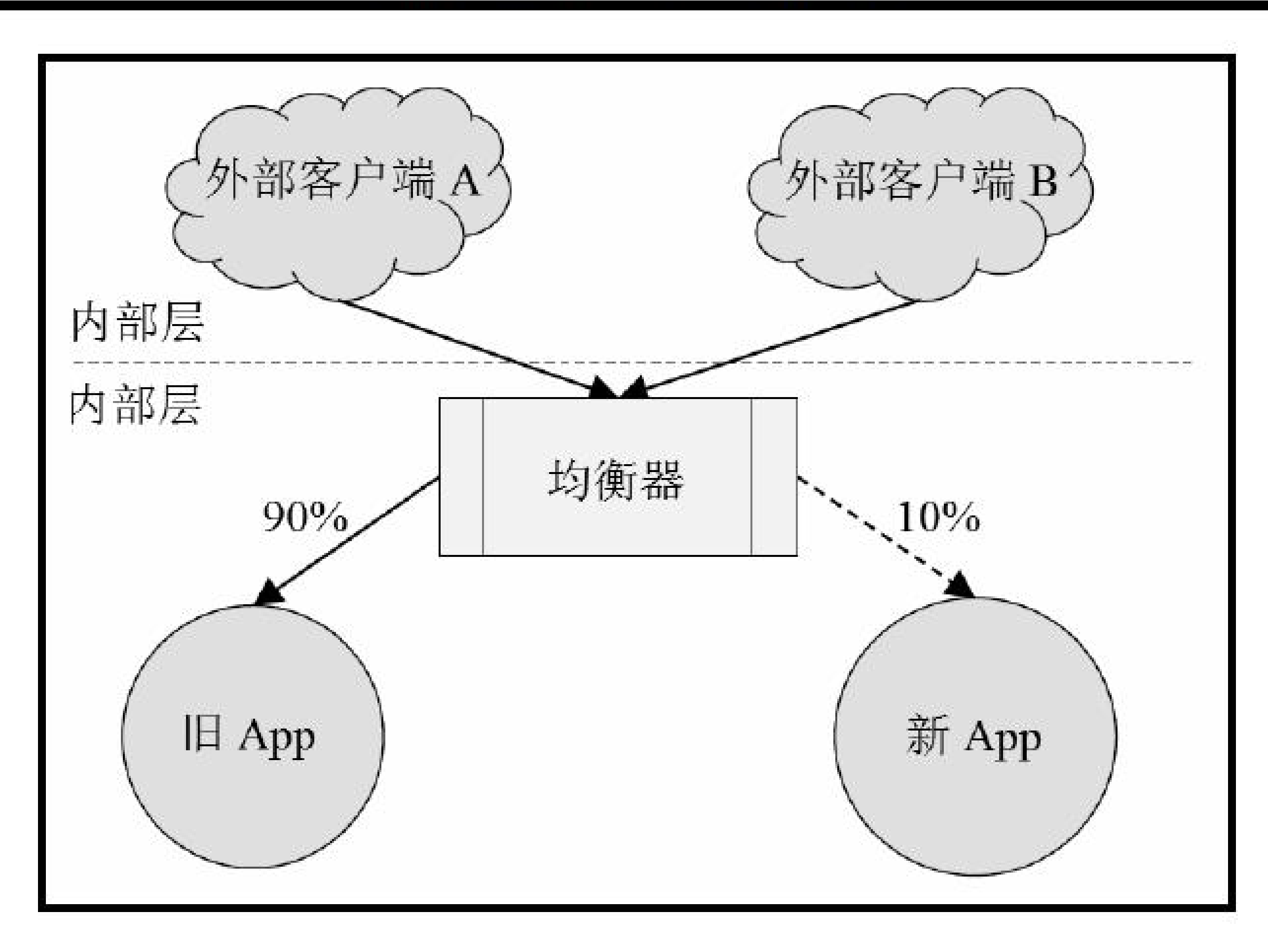


图 1.9

替换处理过程所涉及的概念十分简单。利用控制作为平衡层，将管控 90%的旧微服务请求，以及 10%的新微服务请求，进而监视和分析新应用程序的成熟度、是否遗忘了某些特性或产生了某些不必要的副作用。

这种方法减少了对产品的负面影响，并提供了关于新应用程序的真实数据。随着新的微服务不断成熟以及对特性可信度的提升，将对新的微服务发布更高比例的请求。更为重要的是，鉴于小型业务范围和低耦合，微服务很容易被替换。在业务和堆栈方面，完全替换服务都将是一个较为自然的处理过程。

1.3　轻量级通信

在单体系统中，许多项目仅仅因为通信层中的问题而无法成功地迁移到微服务体系结构。当然，当我们讨论容器、分布式应用程序和业务域划分时，读者可能会对某些术语感到陌生，如延迟和数据转换。

单体应用程序中的通信由内部组件构成，如方法、函数、属性和参数。在该生态系统中，延迟和数据转换彼此无关。在微服务中，则需要对此加以全盘分析。

微服务之间的通信涵盖以下两种方法：

- 同步方法。
- 异步方法。

读者应理解每种形式的工作方式，如表 1.1 所示。

表 1.1

	一对一	一对多
同步方法	请求/响应	
异步方法	通知	发布/订阅
	请求/异步响应	发布/异步响应

在表 1.1 中，所用的通信类型将根据域需求而变化。对于直接和顺序系统，同步通信更加适宜。在不需要即时响应的任务中，异步方案则更为可取。

1.3.1　同步

对于微服务之间的通信，同步方法则是一种应用更为广泛的方案。其中，某些协议较为知名，而其他一些协议则较少被提及。直接协议的范围包括以下内容：

- HTTP。
- TCP。
- WebSockets。
- Sockets。
- RPC。
- SOAP。

一般来讲，最常用的实现是 HTTP 协议。许多微服务使用 HTTP 进行通信，因为 HTTP 通常与 JSON 一起使用。

该方案的问题在于，当采用 HTTP 协议时，JSON 在发送和传输信息时，其处理时间往往难以令人满意。对于 App-App 通信和 API 常规连接，一些采用 JSON（基于 HTTP 协议）的团队仅采用了维持策略。

对于 HTTP，基于 JSON 的 API 在实际操作过程中可视作一种规范行为；而在微服务间的内部通信中，这将会带来一定的问题。考虑到延迟和数据传输问题，一种较好的方法是针对微服务间的通信使用二进制传输。

这种方法涵盖多种选取方案，例如 Avro、基于 CPRM 的协议缓冲区和 Thrift 等。另外需要注意的是，对于二进制，我们不需要绑定任何特定的技术，使用该方法更改通信接口非常简单。

1.3.2　异步

在微服务间的某些直接通信中，时序问题非常重要。但在一些场合下，异步处理已然可满足要求，且不需要立即响应或确认成功，只需要简单地运行任务即可。对于这种方法，消息代理可完美地解决这一类问题。

某些软件应用程序对消息代理提供了较好的支持，如 RabbitMQ、ActiveMQ、ZeroMQ、Kafka 和 Redis。每一种选择方案都拥有自己的特点，如速度、弹性。同样，业务设置将决定使用哪一种技术。

1.4　异质/多语言

在软件开发中，并不存在单一的解决方案，此言极是。对于经过良好整合的微服务，一种有趣的现象是，微服务间可能采用了完全不同的技术。针对某个问题的解决方案，微服务的这一异质特征使得团队能够寻找最为适宜的处理方法。

当使用多语言的微服务时，对于部署和维护，理解其不断增长复杂度是十分重要的。然而，异构应用程序对此提供了有益的补充。

1.5　通信的文档化

团队之间的通信相对复杂，无论是技术团队还是业务团队。在其他时候，各种团队之间的技术通信（如前端、后端和移动端）可能代价高昂，某些交付、提交或特性功能往往会被延迟。顺畅的通信机制对于任何项目的成功都是至关重要的。

在内部代码或简单的文档中，编写良好的文档是团队间知识标准化的最佳方式。针对这一问题，Swagger API 可视为一种较好的替代方案，如图 1.10 所示。

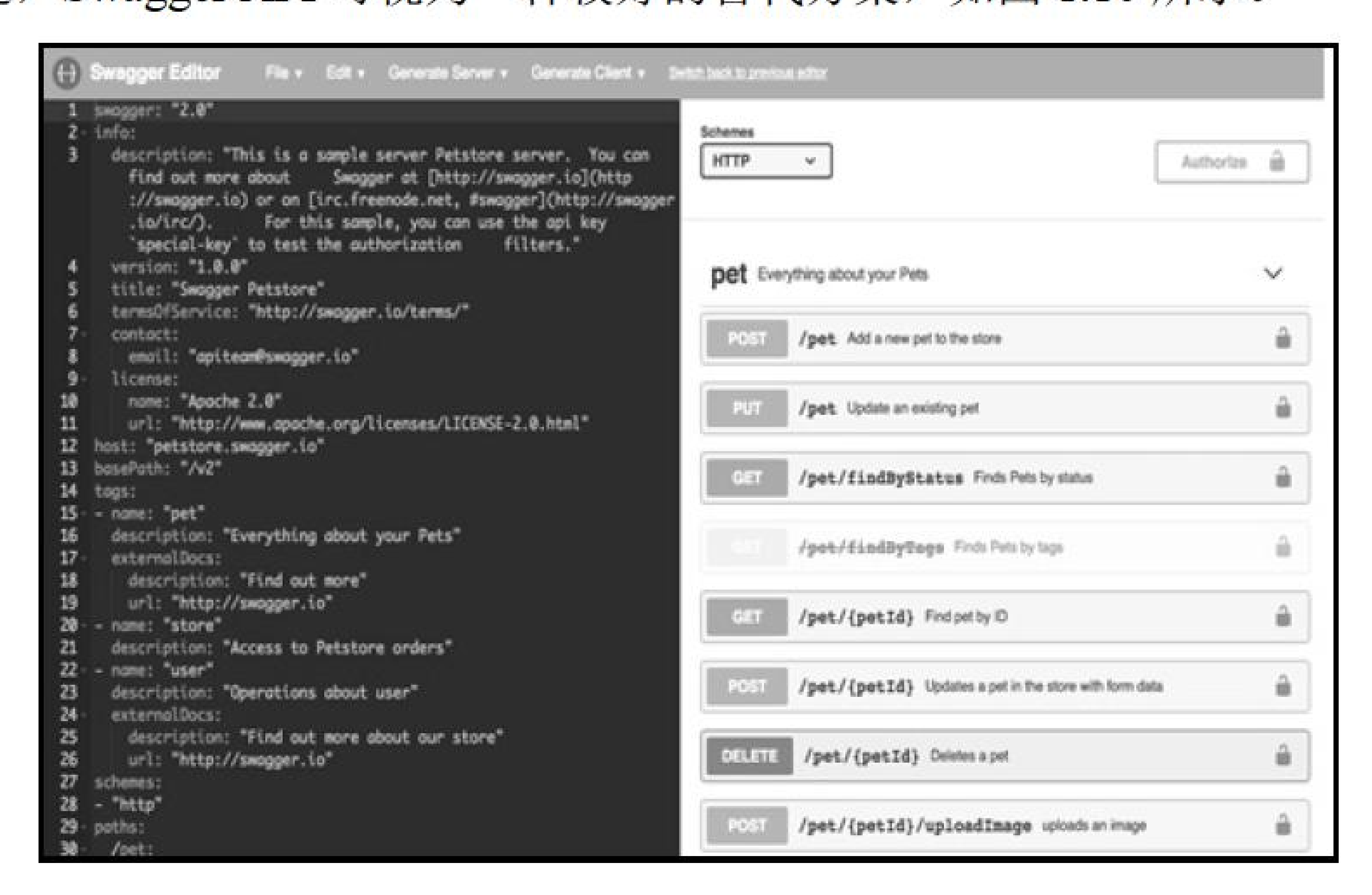


图 1.10

通过简单的配置文件，全部 API 均可实现团队（开发需求团队和开发解决方案团队）间的集成开发。

1.6　Web 应用程序端点

在前述发布接口中曾谈到，定义端点的内容及其尺寸十分重要。

一个较为重要且尚未被解决的话题是，处于公开状态的端点的组件化。考察与用户信息相关的微服务，某些开发团队决定构建大型端点，进而提供与用户相关的全部信息。这种类型的端点（getUser 类型）易于开发，但却难以扩展。

对于那些使用 API 的用户来说，大量无用的信息可能正在被传递，或者是需要传输的信息较为特殊，且由微服务生成的开销较大。实际上，较为明智的方法是创建一个更加分散和多样化的信息 API，如果需要使用 getUser，则创建一个包含较小信息的编排器，并在单一端点上传递，如图 1.11 所示。

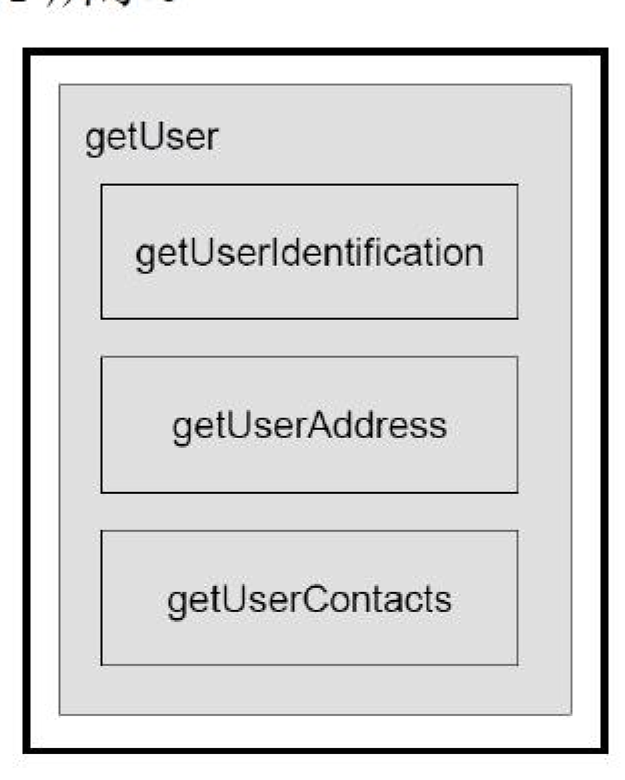


图 1.11

这一策略类型称作端点构造器，其中，重要的信息实际上是其他轻量级数据源的组合。

1.7　移动应用程序端点

诸如速度和加权信息这一类问题在 Web 中并不十分常见，但在移动环境下，情况则有所不同。

为移动应用程序提供轻量级和成熟的 API 的组件化端点，对于业务成功至关重要。没有人希望电量被耗尽，只是因为应用程序包含开销巨大的端点。

在移动生态系统中，前述内容提到的基于 getUser 的 API 是完全不切实际的。微服务限制的定义不仅涉及构成微服务域的内容，而且还包括公开该域的数据。

1.8　缓存客户端

对于 Web 应用程序来说，缓存策略是重点讨论的内容之一；而对于微服务，情况也大致如此。

当谈及客户端上的缓存时，一般意味着，如果确实必要，请求只传递到后端进行处理。换而言之，对于近期已经实现了的请求，这将试图阻止直接访问后端。

针对该策略，一种十分有效的工具是 Varnish Cache。Varnish Cache 加速器是一个 Web 应用程序，也称作反向 HTTP 代理缓存。图 1.12 显示了 Varnish Cache 中的相关操作。

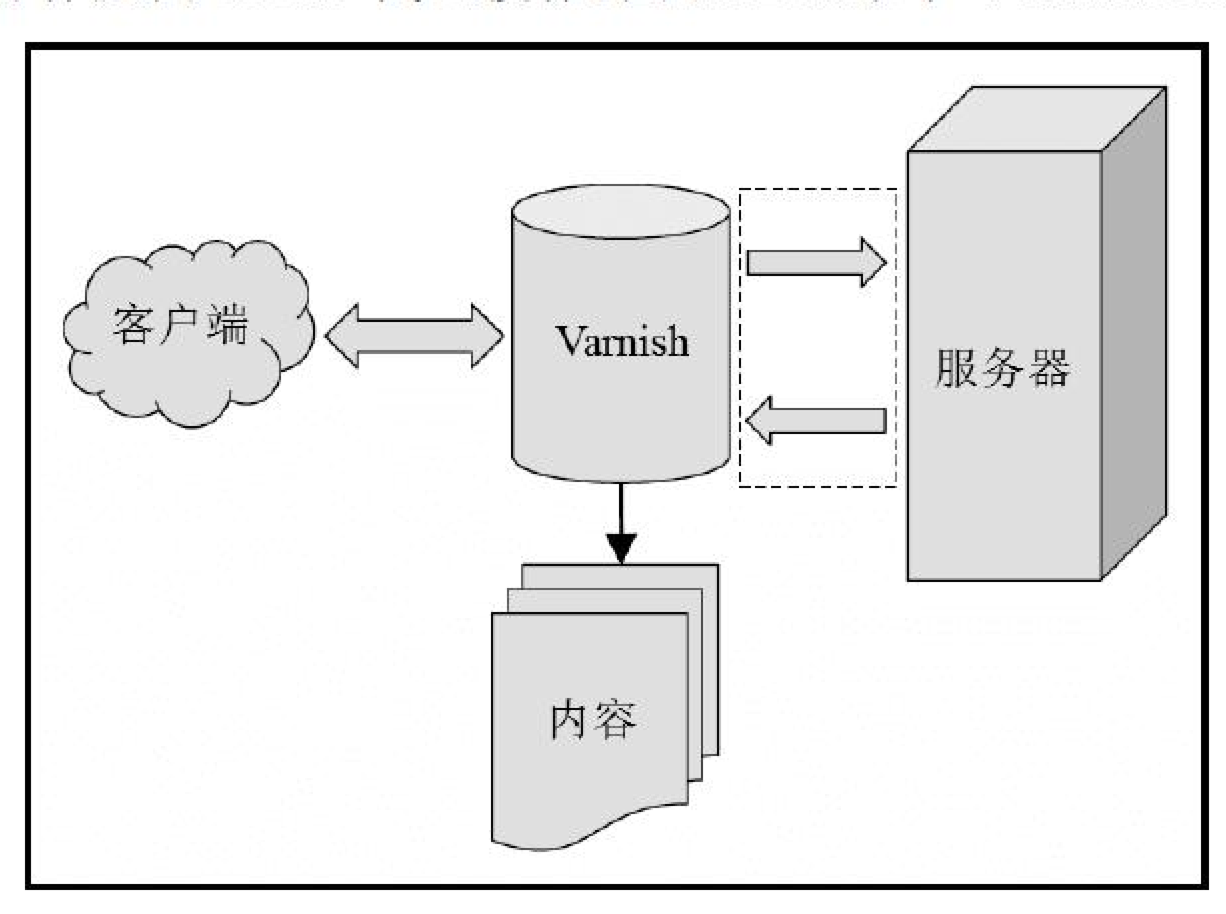


图 1.12

请求来自各种类型的 Web 客户机。Varnish Cache 仅在首次时将请求传递给服务器，并存储来自服务器的接收数据。如果对 Varnish Cache 中已经存在的相同信息进行第二次请求，那么该缓存将回答请求，以使服务器免受此类访问打扰。

Varnish Cache 可以在内存中存储许多不同类型的信息，但重要的一点是，其目标是传输的数据。如果信息不是组件化的，Varnish Cache 总是让请求进入服务器；用户无法知道请求是否是相同的类型。

1.9　调节客户端

设计良好的微服务具有高度的可扩展性，但它确实意味着可拥有无限的资源。在云计算资源有限的情况下，情况则有所变化，提供服务的成本可能会变得非常高，以至于阻止了相同的服务。

考虑到这一点后，我们可以采取一些措施来降低开销成本。正如前面提到的，高效的缓存的实现便是其中之一。然而，这还不是全部内容：某些时候，为了阻止资源的高消耗，调节（节流）行为也不可或缺。

下列情形缺少一定的可行性：作为 Web 页面，微服务的客户端针对当前微服务运行大量请求；或者同一页面尚未完善，且无法处理已经接收到的数据。

为此，简单的调节机制对于减少微服务的消耗非常有效，即保持指向信息使用以及传输至客户端的引用。

下列内容涵盖了一些相关的调节策略。

- ❑ 来自同一客户端的每分钟请求数。
- ❑ 来自同一客户端的每秒请求数。
- ❑ 对于类似信息，来自同一客户端的每分钟请求数。
- ❑ 对于相同信息和客户端，每秒的请求数。

据此，即可抑制某些潜在的错误，如不适当的数据操作、不可靠的 Ajax 请求，以及相对简单的攻击尝试行为。

1.10　确定贫血域

若缺乏成熟的业务层以实现自身的任务，那么，此类微服务可视为构建于贫血（anemic）域上的软件示例。

对此，可以通过以下简单的观察来识别贫血域：

- ❑ 微服务无法仅使用接收到的数据执行自身任务。
- ❑ 微服务需要获取多个端点中的数据执行某项任务。
- ❑ 微服务缺乏自给自足的实体模型。
- ❑ 微服务在另一个微服务中等待任务完成，以执行后续操作。
- ❑ 微服务需要利用其他外部微服务共享资源；此类资源可缓存至样本数据库中。

如果正在开发的微服务具有上述特征之一，那么它可能是一个薄弱的领域。如果微服务具有所列的两个或多个特征，那么它肯定是一个贫血域。

贫血域对微服务生态系统是非常有害的，当纠正由于各自领域成分不足而产生的技术债务时，其开销倾向于成倍增加。

1.11　确定 fat 域

在许多情况下，微服务执行的任务往往会超出其应有的数量。显然，一切都很好，部署也简化了。但事实上，这是一个 fat 域。微服务之所以没有这个名称，不仅是因为它们是一个小应用程序，而且还包含一个小而简单的业务域。若微服务在某个特定领域中包含限制条件，通常意味着该应用程序在初始状态下构建于一个小型单体之上。

针对我们的新闻门户网站，用户可视为微服务的较好的候选者。构建一个管理用户数据的微服务是很有意义的。然而，在单体应用程序中，用户层与 AAA（身份验证、授权和记帐）间一般具有很强的连接。

当涉及微服务数据时，用户和 AAA 之间无须耦合。这主要是因为 AAA 的整体过程不仅限于终端用户，还包括移动、前端和使用者 API 等客户端。在这种情况下，用户微服务代表一个 fat 域。

上述 fat 域的划分可通过两部分内容所持有，即 AAAService 和 UserService。另一种方法是基于网关 API 的 AAA 职责。对于产品的整体增长来说，基于这些独立领域的功能可扩展性和实现特性则要有趣得多。

对于最终产品的增长和可扩展性，理解域的尺寸和限制条件十分重要。

1.12　针对业务确定微服务域

现在是理解本书中开发的业务领域的时候了。具体来说，域包含在我们的单体应用程序中，下面回顾一下它是如何构成的。这里的单体 Django 由以下 3 个 Django 应用程序组成：

- ❑ News。
- ❑ Recommendations。
- ❑ Users。

需要注意的是，在当前上下文中，鉴于 Django 的设计方式，用户和 AAA 处于耦合

状态。我们已经看到，对于微服务来说，这并非是一种理想状态。

另一点需要注意的是，新闻不一定会产生单一的微服务；我们可以根据不同的新闻类型创建不同的微服务。针对每种不同类型的新闻内容，这将促进 API 的目标定位和可扩展性。假设当前门户网站中涵盖了体育、政治和名人新闻等内容。如果开发了一个新的主题，那么，将为这个主题创建一个新的 News 微服务。这种方法为应用程序的这一部分内容提供了类似于 Z 轴的可扩展性。

首先，当前域将划分为以下类别：

- SportNewsService。
- PoliticsNewsService。
- FamousNewsService。
- RecomendationService。
- UsersService。
- AAAService（可选）。

当然，读者还可以添加新域，并删除其他域；另外，限制当前微服务的视图是我们的主要目标。

1.13　从域到实体

当设置了应用程序中的域后，下面将对实体加以定义。当我们谈到实体微服务时，必须注意，微服务之间的任何事务需求都可能导致设计错误。

代理的进程异步消息可以用于清理数据库，但这并不意味着其间存在事务。尝试在完全分离的微服务之间建立一种事务类型可能是一个很大的错误。当前，我们的旧应用程序包含以下实体：

- 新闻。
  - ID：UniqueID。
  - 作者：FK user_id。
  - 标题。
  - 描述。
  - 内容。
  - 标记：新闻主题。
  - 类型：行为类型（体育、名人新闻、政策）。
  - CreatedAt。

  - UpdatedAt。
  - PublishedAt。
- 推荐系统。
  - Label。
  - user_id
- 用户。
  - ID。
  - 名称。
  - 电子邮件。

除了上述实体之外，为了提供访问权限以及其他权限，还可定义一系列的表以补充用户的信息。

随着微服务架构的整体体系结构的转变，这些实体的数据模型和设计也将发生变化。

首先，我们知道所有的新闻片段都不会是独一无二的。这意味着需要执行类型的删除操作。对于新闻服务来说，其中涉及以下内容：

- ID。
- 作者。
- 标题。
- 描述。
- 内容。
- 标记。
- CreatedAt。
- UpdatedAt。
- PublishedAt。

另一个变化是，用户将不再负责身份验证和授权方面的工作。

1.14　本章小结

本章的目标是从领域和技术角度，向读者展示可扩展微服务架构的基础知识。

我们介绍了一些相关主题，如在微服务中使用域驱动设计、扩展立方体、单职责原则和已发布的接口，以便提供所需的最低限度的理论知识，因此读者可以在接下来的章节中将其投入至实际应用中。

第 2 章将利用所学的各种概念定义堆栈。

第 2 章　微服务工具

在选择微服务堆栈时，尚存在一些争议问题，主要体现在性能、实用性、成本和可扩展性方面，其中大部分内容与背景观点有关，其正确性也值得进一步商榷。

显然，针对堆栈和实现，在指定技术决策时都应该考虑开发团队的历史。然而，在开发一个产品时，有时也需要留下一些舒适区（comfort zone）。舒适区可以是编程语言、协议、框架或数据库，它们可以限制开发人员快速移动的能力。随后，开发的应用程序将变得越来越具可扩展性。

本章将考察内部讨论机制和开发团队方面的内容，重要的是要理解开发堆栈是一项严谨的行为；为了开发最好的产品，我们应该始终考虑成本和可扩展性问题。

本章中的一些批评和建议并非是针对所用技术进行褒贬，全部分析工作均围绕本书的最终产品而展开，即基于微服务的新闻门户网站。

本章主要涉及以下内容：

- 编程语言。
- 微服务框架。
- 二进制通信。
- 消息代理。
- 缓存工具。
- 错误提示工具。
- 区域证明。

2.1　编 程 语 言

讨论编程语言往往会引起争议，主要是因为许多开发人员并未深入地理解编程语言。然而，编程语言应该被视为真正的工具。每个工具都有特定的用途，编程语言也不例外。

本章主要分析新闻门户网站的业务规则，其中将会涉及编程语言的选取。

微服务的一大优点是应用程序的异构性。换句话说，不需要考虑将一个堆栈应用到所有业务领域。因此，我们可以定义所用的每个微服务堆栈，包括所涉及的编程语言。

基本上讲，任何满足互联网的编程语言都可以在微服务中使用，其中的差异源自编

码的需求条件和域边界。

如果微服务具有很强的数学处理负载需求，或者数值的不可变性占据主导地位，那么函数语言将是一种较好的方式；如果需要处理大量的数据，那么答案可能是使用基于虚拟机的编译语言。

否则，项目的交付日期，甚至是应用程序的整体架构均会受到影响，因此，在进行定义之前，需要对以下几方面内容加以分析：

- 熟练程度。
- 性能。
- 开发的实用性。
- 生态系统。
- 扩展的开销。

2.1.1 熟练程度

软件开发人员的第一个目标是编程语言或范例的熟练程度。达到良好的熟练程度并不容易，有些语言的学习曲线可能比其他语言更加陡峭。

当熟练使用一种语言后，最终会产生一个舒适区，开发人员或团队发现很难离开时，问题也会随之出现。相反，必须打破一个神话：一种编程语言比另一种要容易得多。或许，一种语言在开始阶段可能会相对简单，但最终的结果取决于开发人员所经历的实践时间，以及所面临的各种处理场合。

另一个需要改变的想法是，所有语言其核心内容都是平等的，只有语法才会改变。然而，尽管语言之间具有相似的语法，但在内部设计和性能上，它们是完全不同的。

对于微服务来说，当考虑使用哪一种编程语言时，熟练程度确实是不可忽略的因素，但也并非是决定性因素。

2.1.2 性能

当为微服务选择相关语言时，性能可视为一个较为关键的需求条件。当探讨微服务性能时，可能会涉及多种问题，例如网络通信层、数据库的访问等。所有这些对于微服务来说都是关键之处。因此，编程语言的执行速度不能过于缓慢。

当目标致力于微服务的性能时，无论开发团队的技能如何，它应该被用于最好的第二语言基准测试和压力测试。

一种误解是，一味地追求开发团队实现某个特性，以及性能需求的速度。性能与度

量标准有关，类似于代码响应请求或执行任务时的行为。当然，个人或团队绩效不包括在这一度量标准中。

2.1.3　实践开发

该需求条件负责测量产品特性的投放速度。实践开发涉及两个团队：已经存在的开发团队和可能存在的开发团队。

如前所述，“成功”一词对应用程序和产品所有者来说都意味着问题的出现。保持代码简单易懂是修改代码和实现新特性的基础。

良好的编程实践有助于我们对遗留代码的理解，但通常仅限于语言自身，其原因在于，冗余内容往往缺乏友好性。

在某些情况下，一种编程语言由于其自身的特性是极具表现力的。但是，实现新内容的时间成本可能非常昂贵，尽管其过程可能很简单。

设想一下，一家初创企业刚刚推出了自己的产品，该产品是一种最小化可行性产品（MVP），在投放市场后执行一般的公开化验证。如果 MVP 获得成功，那么，须尽快发布新的特性。在这种情况下，代码新增交互行为中的实用性则是当前的主要问题（而非性能）。

因此，当开发微服务并决定使用某种语言时，上述问题应引起足够的重视。

2.1.4　生态圈

编程语言的生态系统是一项至关重要的内容。

我们都知道，在开发任何应用程序时，框架的某些部分对于速度和简单性几乎是必不可少的。对于微服务，情况也大致相同。

可以这样讲，新特性的研发并非是技术屏蔽所造成的。当然，微服务体系结构提供了多个非常广泛的工具选项。当选择某种编程语言时，了解其可能存在的缺陷，并继承它的生态系统，对于负责实现的工程团队来说是至关重要的。

在某些情况下，编程语言非常具有表现力，但以开发速度取胜的生态系统往往会对性能产生损害，而这种现象普遍存在。

另一点需要注意的是，当一种语言较为简单，但是框架还不够成熟时，最终将会引入不必要的复杂性。

观察一种编程语言的生态系统，并了解继承带来的风险，是选取一种语言时须考察的基础内容。

2.1.5　扩展性的开销

扩展应用程序的成本与两个主要因素有关。第一个因素是实现软件时所选堆栈的速度，特别是指算法处理的速度和能力，以及请求响应。第二个因素则是与业务应用相关的扩展能力。例如应用于特性上的时长，特别是新特性的可预测性。另外，创建新内容或对现有内容重新设计均是一项十分耗时的工作。

在微服务体系结构中，扩展性的开销通常与较小的域和较少的集成部分这一类概念有关。即便如此，这一类开销也是非常重要的。

下面考察两个应用程序，第一个应用程序与终端用户具有很强的交互性，例如在线游戏或实时编辑文档。第二个应用程序则仅具备展示特征，并设置了一个编辑区域，但不会向所有用户开放，例如报纸或流媒体提供商。

实时数据处理应用程序，以及对请求的响应时间须具备快速、动态的特征。在第二个应用程序中，鉴于缺少相关性，因而信息可位于缓存或静态存储中。

如果读者未深入理解微服务的本质，那么，所付出的代价可能相对高昂。

2.1.6　选取编程语言

综上所述，下面将通过相关知识来选择微服务的每个领域中所使用的编程语言。

当前新闻门户网站包含以下领域：

- ❑ SportNewsService。
- ❑ PoliticsNewsService
- ❑ FamousNewsService。
- ❑ RecommendationService。
- ❑ UsersService。

当给定业务领域后，可通过特征相似性对其进行划分。其中，SportNewsService、PoliticsNewsService 和 FamousNewsService 包含类似的行为。此类微服务表示为新闻提供商，更多地强调数据的使用，而非接收信息。

这些微服务可能包含一个相同的起始堆栈，但这并不意味着它们始终保持一致，抑或按照相同的方向发展。对于编程语言来说，性能并不是那么重要，变化的速度和新特性的实现才是至关重要的。

RecommendationService 不同于其他微服务，终端用户和该微服务之间不存在直接交互；另外，编辑区域与该软件之间也不存在直接交互。RecommendationService 是一类支持型微服务。除此之外，还包含了其他一些微服务，全部交互行为和操作均在技术层面

上进行。相应地，加载和交互行为完全以异步方式出现，但与其他微服务相比，其处理开销相对较高，毕竟这不是一个实时应用程序。

针对终端用户和编辑层，UserServices 微服务均包含了动态交互行为。源自 UserServices 的信息也可供其他微服务使用，如 RecommendationService。需要注意的是，在该层中，缓存行为比较危险，如果未实施正确、有效的操作，将会提供错误的信息。由于 UserService 微服务支持与互联网之间的直接关系，对应的编程语言应具有请求响应速度、处理速度、实现特性的简单性和良好的异步 API。

考虑到各部分内容的特性，下面将选择一种适用于每个微服务的编程语言，并具体考察上述新闻门户网站中的 5 个领域。

编程语言间的比较过程相对复杂，此处我们需要对其做出选择。许多语言是可以进行比较的。然而，对于当前应用程序，我们根据流行程度、个人经验、文档和实际的适用性选择了 5 种编程语言进行比较，其中包括 Java、C#、Python、JavaScript 和 Go。

1. Java

Java 完全符合面向对象范式，且非常具有执行力，同时降低了扩展的成本。随着 Java 语言的发展，该语言不再像以前那样冗长，但仍然缺少应有的简洁性，要求开发人员非常熟练地维护代码和实现新特性。在生态系统方面，Java 虚拟机非常出色、成熟，而且十分稳定，但是框架通常较为复杂。

2. C#

与 Java 类似，C#完全符合 OOP 范式且极具执行力，同时也降低了扩展成本。C#与 Java 具有相似的一面，且兼具一些额外的实用性特征。此外，C#对开发人员的熟练程度要求较高，进而可保证开发速度。C#的生态系统是非常成熟的，对应的框架也不是那么复杂。

3. Python

Python 并不具备 Java、C#和 OOP 的语法特性，但它很好地迎合了范式。作为一种解释型语言，其性能并不如之前提到的几种语言，这意味着需要更多的服务器来支持解释型语言所不支持的相同负载。然而，Python 语言的简单性表明，当涉及代码维护和新特性开发时，扩展行为的高成本得到了补偿。开发人员只需在较短的时间内即可掌握这门语言。另外，Python 生态系统中包含大量的简单框架。在同一生态系统中，语言解释器涵盖一系列的选项，这对性能的提升有很大的帮助。

4. JavaScript

毫无疑问，JavaScript 是在前端和后端之间产生较少冲突的一种语言。对于后端来说，

JavaScript 是可选的；而在前端中则具有一定的强制性。开发人员需要深入理解 JavaScript 的内部行为，以避免可能出现的错误。相应地，最适用于 JavaScript 的范例是函数式的。另外，JavaScript 生态系统中存在大量的框架。在某些情况下，框架数量众多也使问题变得相对复杂。JavaScript 的性能表现优异，但是需要熟练地维护代码和编写新特性。

5．Go

Go 是一种编译的编程语言，且具有较好的性能。毫无疑问，它是近年来越来越受欢迎的语言之一。Go 是一种基于命令式范式的语言，同时也包含了某种层次的 OOP 特征——较少程序员对此有所了解。Go 语言生态系统的主要特征是一个标准库，在某些情况下，框架变得不再必需。但是这个生态系统并不完美，其中包含了诸如版本控制这一类简单的问题。Go 具有简单且易读的语法。其主要特性体现在应用的方便性以及并发编程的处理方式。

为了更清楚地进行比较，图 2.1 显示了各种语言所支持的各项功能。

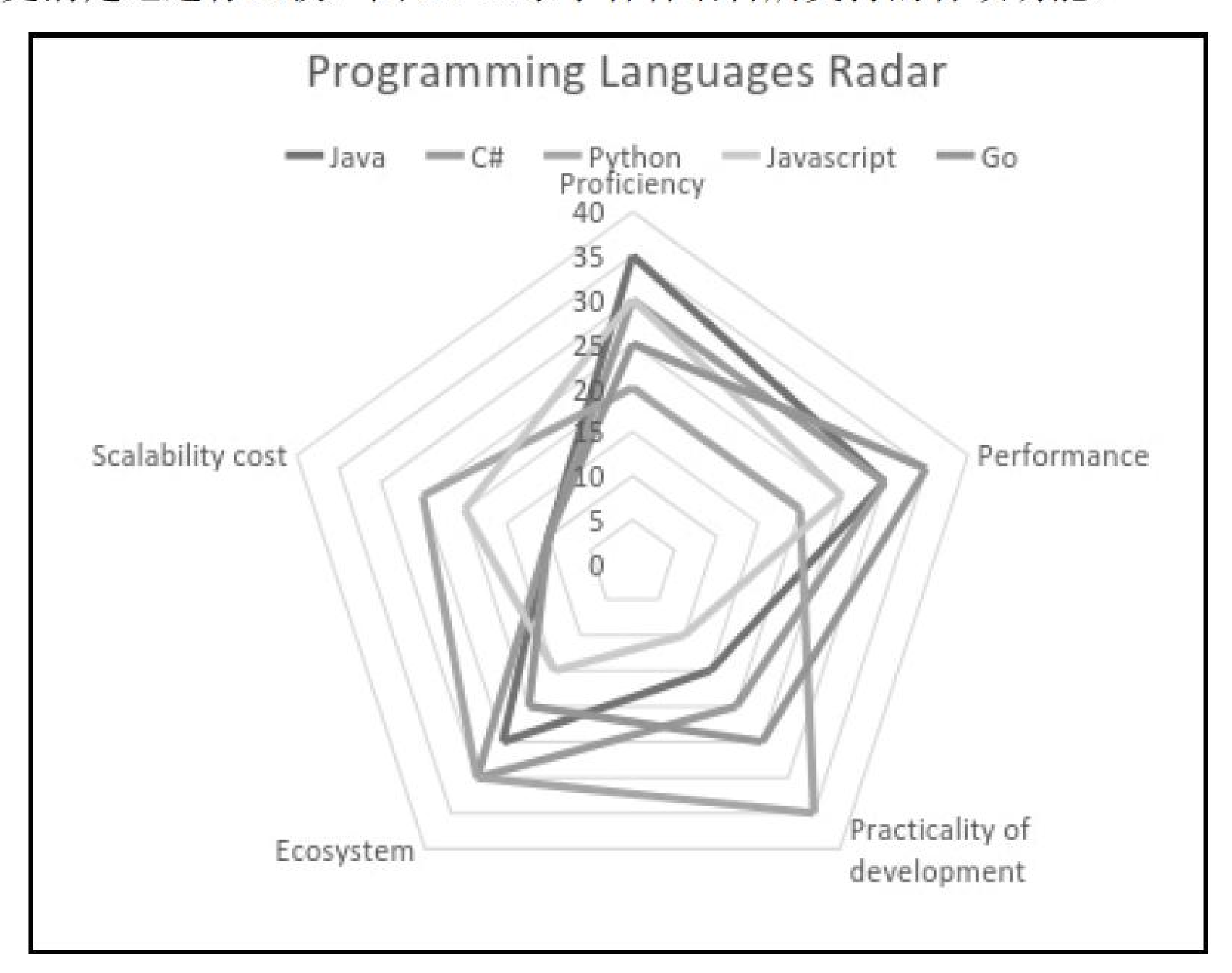


图 2.1

在我们的新闻门户网站示例中，所采用的编程语言包含以下内容。

- SportNewsService、PoliticsNewsService 和 FamousNewsService：此类微服务采用了 Python 语言，这也是该语言的用武之地。即使门户网站的访问量较大，性能

稍逊一筹的 Python 语言也不会产生任何问题。

- RecommendationService：该应用程序同样使用了 Python 语言，这一选择结果与使用其他工具时性能和实用性之间的关联有关，进而构成我们的微服务栈。此类微服务并不要求具备实时特征。我们可以使用一些简化的 API，且不会对生态系统的其他部分造成破坏。
- UserService：该微服务使用了 Go 语言。UserService 与用户进行交互，同时向其他应用程序微服务提供信息。

实际情况是，没有一种工具可以做到尽善尽美，每种语言都包含了自身的优缺点。在微服务架构可用的众多技术中，当前任务是对适用于各种场合下的堆栈进行管理。

2.2　微服务框架

当对框架进行处理时，考虑到框架技术的多样性，我们将至少使用 3 种框架以维护相应的生态系统。

当然，我们可以将所有的微服务放在同一个堆栈中。但是，为了搜索每个领域的最佳整体性能，我们选择了一个更复杂的堆栈。

显然，在开始阶段，复杂度将超出预期。但这种复杂度与每种情况所体现的性能相匹配。

基本上讲，新闻门户网站选取了 3 种不同的编程语言，即 Python、Go 和 C#。下面考察针对每种语言所采用的框架。因此，在为每个微服务构建开发堆栈方面，我们又向前迈出了一步。

2.2.1　Python 语言

Python 中包含了多种框架，如 Bottle、Pyramid、Flask、Sanic、JaPronto、Tornado 和 Twisted。除此之外，Django 也是一种获得广泛支持的框架。

这里，Django 框架用于构建我们的新闻门户网站，并使用了单体版本。对于全栈应用程序来说，它是一个非常好的工具，且安装起来十分简单。然而，对于微服务来说，个人认为它并非是针对这一任务的最佳框架。首先，API 的公开化并不是 Django 的固有特征。使用 Django 创建 API 来自 Django Framework Rest 应用程序。

为了维护 Django 框架的标准结构，简单的 API 设计稍显不足。Django 为开发人员提供了整个开发堆栈，在开发微服务时，这并不总是用户期望的方式。更简单和更灵活的

组合框架可能更加实用。

Python 生态圈中还包含了其他工作框架，对于 API 而言同样工作良好，Flask 便是其中之一。像“Hello world”这一 API 可通过简单的方式生成。

读者可利用下列命令安装 Flask：

```
$ pip install flask
```

另外，端点的编写方式也十分简单，如下所示：

```
# file: app.py
# import de dependencies
from flask import Flask, jsonify
```

下列代码显示了框架的实例化操作：

```
app = Flask(__name__)
```

例程的声明如下所示：

```
@app.route('/')
def hello():
    #Prepare the json to return
    return jsonify({'hello': 'world'})
```

主程序的执行方式如下所示：

```
if __name__ == '__main__':
    app.run()
```

上述代码将以 JSON 格式返回“hello world”消息。

Flask 也引出了其他一些框架，如 Sanic，其简洁性和语法与 Flask 十分相近。

利用下列命令可安装 Sanic：

```
$ pip install sanic
```

Sanic 的语法几乎等同于 Flask，如下所示：

```
# file: app.py
```

下列代码用于导入依赖关系：

```
from sanic import Sanic
from sanic.response import json
```

Sanic 的实例化操作如下所示：

```
app = Sanic()
```

例程的声明如下所示：

```
@app.route("/")
async def test(request):
   #Prepare the json to return
   return json({"hello": "world"})
```

下列代码将执行主程序：

```
if __name__ == "__main__":
   app.run(host="0.0.0.0", port=8000)
```

性能是 Flask 和 Sanic 之间的最大差异。鉴于采用了 Python 中的新特性，以及 Sanic 框架的交换能力，Sanic 框架的性能优于 Flask。然而，从成熟度和市场表现来看，Sanic 尚不如 Flask。

Flask 的另一个特性是与其他工具的集成，如 Swagger。据此，不仅可编写 API，还可实现较好的文档化管理。

对于采用 Python 作为编程语言的微服务，实用性则是至关重要的问题，而非性能，同时将该框架用作 Flask 微服务。

2.2.2　Go 语言

Go 语言则与 Python 完全不同。大多数 Python 框架对性能均有所帮助，并试图为开发过程提供最佳环境，但 Go 是不一样的。通常，包含许多特性的框架往往会影响语言的性能。

1. 日志

日志方面则无太多变化，读者可以参考 logrus 文档（对应网址为 https://github.com/Sirupsen/logrus）。它是一个非常成熟和灵活的 Go 日志库，其中包含了用于不同工具的钩子程序，如 syslog 和 InfluxDB。

加速记录日志并提升实现的可行性则是 logrus 的主要特征。

2. 处理程序

在 Go 中创建路由 API 非常简单，但是本地处理程序的选项会带来某些复杂性，特别是关于验证方面的问题。本地合成器缺少一定的灵活性，所以最好的选择是寻找更高效的工具处理程序。

对于处理程序，Go 语言包含多个选项，而且由于底层语言的编写特性，它可能是使用最多的库模型。

对于 Go 语言中的路由性能，在本书编写时，Fasthttp 的表现十分抢眼（对应网址为

https://github.com/valyala/fasthttp）。Fasthttp 库采用 Go 语言编写并提供了底层语言。此外，Fasthttp 中的各项指标其表现均较为突出。

下列代码针对所提供的静态文件在本地进行测试：

```
Running 10s test @ http://localhost:8080
 4 threads and 16 connections
 Thread Stats   Avg      Stdev     Max   +/- Stdev
   Latency   447.99us   1.59ms  27.20ms   94.79%
   Req/Sec    37.13k    3.99k   47.86k    76.00%
  1478457 requests in 10.02s, 1.03GB read
Requests/sec: 147597.06
Transfer/sec:    105.15MB
```

其中可看到，每秒的请求数量超出了 140000 个，但是，当采用 Fasthttp 编写路由时，其复杂程度与本地库相当。鉴于这一问题，某些框架可针对 Fasthttp 定义接口，fasthttprouter 便是其中之一（对应网址为 https://github.com/buaazp/fasthttprouter），最终会创建许多开发构件，而不会过分损害 Fasthttp 的良好性能。

编写性能良好的路由方案是我们所追求的目标，但也需要在性能与稳定性之间寻求某种平衡。Fasthttp 及其所有辅助接口都修改了处理程序的本地标准以实现上下文本身。如果存在较为严重的性能问题，可能需要考虑使用 Fasthttp。当前尚不会面临这一类问题。因此，建议使用与标准 Go 接口更兼容的内容。

gorilla/mux 是较为知名的选项（对应网址为 https://github.com/gorilla/mux）。毫无疑问，对于 Go 语言来说，这也是最为成熟和应用广泛的库。

3. 中间件

对于中间件组合，可使用 Negroni，对应网址为 https://github.com/urfave/negroni。Negroni 除了是一种十分成熟的工具之外，还与 Mux Gorilla 以及本地 API Go 完全兼容。

4. 测试

对于单元测试，可使用 testify，对应网址为 https://github.com/stretchr/testify。testify 是一类较为简洁的库，并形成于断言和 Mock 中；而对于功能测试，则可采用默认的 Go 语言库。

5. 包管理器

如果说 Go 语言生态系统还存在问题的话，依赖关系管理便是其中之一，该问题应引起足够的重视。

顺便提及一下，Git 是 Go 依赖关系的官方存储库，这里涉及全部 Git，无论是 GitHub、Bitbucket 还是其他内容。此处的问题是，当使用 Go 命令（go get…）下载依赖项时，应

用程序的版本总是主存储库中的版本，因而尚缺少严格的添加控制行为。

包管理器将使用到 godep，对应网址为 https://github.com/tools/godep。godep 是一个简单的工具，用于控制项目中所使用的版本，并利用存储库 URL 和散列 Git 历史保护 JSON 文件。

6. Golang ORM

Gophers 的一个特征是使用 Go 给开发者命名，而不是使用 ORM。通常情况下，首选项仅使用数据库中的通信驱动程序。

Gophers 一般不使用 ORM，而是通过某个工具，在 Go 结构中生成更加实用的数据库信息。这种关系数据库的工具是 SQLX（对应网址为 https://github.com/jmoiron/sqlx）。

SQLX 的工作方式与 ORM 不同，它只是为本地 Go 包创建一个更友好的接口，以便与数据库/SQL 进行通信。

如果选择的数据库应用程序是 NoSQL，那么将很难采用任何数据解释工具，因为最实用的方法是仅使用有效的驱动程序。

2.3　二进制通信——服务间的直接通信

前述内容讨论了与微服务相关的许多话题，包括协议、层、类型以及包尺寸。

微服务间的通信对于项目的成功至关重要，问题的关键之处在于，如何在不影响产品性能的前提下与终端用户进行沟通。

如果产品的开发和部署缺乏可伸缩性，或者终端用户的体验受到损害，那么，其功能将大打折扣。

关于这个问题有很多文献和研究材料可供读者阅读，尽管如此，挑战仍然存在。开发人员在项目的这一部分内容中常会犯错误。

微服务之间只存在两种形式的通信，即同步和异步方式。最常见的是微服务之间的异步通信，该方式更容易扩展，但问题之处有时也更难以理解。对于微服务之间的同步通信，可以很容易地理解该领域中可能出现的错误，但扩展行为则相对困难。下面将处理同步通信问题。

2.3.1　理解通信方式

第一步是理解微服务的功能，以及哪一种通信方式最为适用，这里以微服务推荐系统为例。微服务推荐系统是一个微服务，但不与客户进行直接通信，而是跟踪用户的个

人资料。同时，不存在其他应用程序从该微服务中获得即时响应。因此，这个微服务的通信模型是异步的。

也就是说，RecommendationService 并非是同步情形。那么，答案又是什么呢？

答案是 UserService。当用户输入与 UserService 通信的既定 API 时，该用户可立刻看到变化。当微服务请求关于 UserService 的一些信息时，我们希望即刻获取最新的信息。因此，UserService 是可以应用同步通信的服务。

但是如何在微服务之间创建一个良好的同步通信层呢？稍后将对此加以讨论。

1. 同步通信工具

微服务之间最常见的直接通信形式是使用基于 Rest 的 HTTP 协议，并传递 JavaScript 对象表示法（即著名的 JSON）。通信可有效地支持 API，并为外部应用提供端点。然而，与性能相比，使用 HTTP 与 JSON 进行通信的成本很高。

首先，对于微服务之间的通信，与创建某种管道，或保持连接处于活跃状态的 HTTP 协议相比，优化操作则更为适宜。这里的问题是对连接超时的控制，相关操作应该不是特别严格；除此之外，还可能会关闭门（door）、线程或包含简单静默错误的进程。该方案的第二个问题是 JSON 发送的序列化时间，一般情况下，这将是一个代价相对高昂的处理过程。最后，还需要将数据包的尺寸发送至 HTTP 协议中。除了 JSON 之外，还应对一些 HTTP 头加以进一步解释，并有可能被丢弃。另外，没有必要详细说明微服务之间的各项协议，唯一的关注点应该是维护一个单独的层来发送和接收消息。基于 JSON 的 HTTP 协议在微服务之间进行通信可能是项目中一个严重的瓶颈。尽管协议的实现具有一定的实用性，但优化过程非常复杂且难以理解，因而意义不大。

许多人可能会建议实现通信套接字或 WebSockets。但最终，这些通信层的定制过程与经典的 HTTP 非常相似。

微服务之间的同步通信层须完成下列 3 项基本任务：

- ❑ 报告实践结果，并对期望的消息进行控制。
- ❑ 发送简单、轻量级的数据包以及快速的序列化操作。
- ❑ 切实维护通信端点的签名。

满足上述需求条件的相关方案将使用二进制或小型数据包进行通信。

当与此类协议协同工作时，需要注意的是，各种方案间彼此不兼容。这意味着，对于序列化和小型数据包的提交操作，工具选择方案应与所有的微服务栈兼容。

市场上一些较受欢迎的选择方案包括：

- ❑ MessagePack（对应网址为 http://msgpack.org/）。
- ❑ gRPC（对应网址为 https://grpc.io/）。
- ❑ Apache Avro（对应网址为 https://avro.apache.org/）。

- Apache Thrift（对应网址为 https://thrift.apache.org/）。

下面考察上述选取方案的工作方式，以及针对新闻门户网站的最佳匹配方案。

2. MessagePack

MessagePack 或 MsgPack 是针对二进制信息的一种序列化器，但是，正如官方工具网站所说，“MessagePack 类似于 JSON，但又速度更快，尺寸更小”。

MsgPack 可快速地序列化数据且尺寸更小，因而针对微服务间的通信提供了高效的数据包。最初，MsgPack 与其他序列化器相比并无太多优势，经过上述修改后，这一问题得到了有效的解决。

当涉及编程语言之间的兼容性时，MsgPack 表现得非常优秀，并拥有市场上最著名的语言库，如 Python 库和 Racket。

MsgPack 在发布的数据包中并未包含本地工具，并将其留于开发人员，其问题在于：仍需要获取微服务间的通信层以支持多语言堆栈。

3. gRPC

与 MsgPack 相比，gRPC 则更加完整，并由数据序列化器 Protobuf 构成，同时使用了 RPC 作为通信服务层。

对于序列化，可创建一个.proto 文件，其中包含了与 RPC 通信相关的信息，必要时还可添加客户端/服务器模型。

下列代码显示了.protocol 文件示例，相关内容通过官方网站工具析取。其中，问候服务的定义如下所示：

```
service Greeter {
```

下列代码显示了问候的发送过程：

```
  rpc SayHello (HelloRequest) returns (HelloReply) {}
}
```

包含用户名的请求消息如下所示：

```
message HelloRequest {
  string name = 1;
}
```

包含问候的响应消息如下所示：

```
message HelloReply {
  string message = 1;
}
```

其中，文件.proto 包含了特殊的书写方式，其优点在于：通信层的签署过程更加标准化，因为在某种程度上，作为序列化模板创建的文件和创建客户机/服务器的文件最终将成为端点的文档。

在生成文件后，如果要创建通信部分，只需运行命令行即可。下面的示例使用 Python 语言创建客户机/服务器：

```
   $ python -m grpc_tools.protoc -I../../protos --python_out=. --
grpc_python_out=. ../../protos/file_name.proto
```

初看之下，上述命令较为复杂，但足以生成通信的客户端和服务器 RPC。gRPC 的发展势头良好，并得到了大量的资金支持。

在兼容性方面，gRPC 无法满足 MsgPack 的相同要求，但与市场上最常用的语言兼容。

4. Apache Avro

Avro 是一个成熟且应用广泛的二进制序列化系统。类似于 gRPC，Avro 也采用了 RPC 作为通信层。

针对序列化处理过程，Avro 使用了.avsc 文件，并以 JSON 格式加以定义。该文件可以由提供了 JSON 的两种类型组成，或者是源自 Avro 自身的更复杂类型构成。

即使作为一种较为成熟的工具，Avro 在与其他语言的本地兼容性方面仍表现得难以尽如人意。考虑到 Avro 项目的开源特征，存在大量的驱动程序可提供与 Avro 之间的兼容性。

5. Apache Thrift

Thrift 是由 Facebook 发布的一个项目，由 Apache 软件基金会负责维护，并与市面上较为常见的编程语言实现了较好的兼容。

Thrift 定义了一个基于 RPC 的通信层，其序列化部分则使用了.thrift 作为模板。.thrift 文件包含了与 C++语言相近的标识符和类型。

下列代码显示了.thrift 文件示例：

```
typedef i32 MyInteger

const i32 INT32CONSTANT = 9853
const map<string,string> MAPCONSTANT = {'hello':'world',
      'goodnight':'moon'}

enum Operation {
  ADD = 1,
  SUBTRACT = 2,
```

```
  MULTIPLY = 3,
  DIVIDE = 4
}

struct Work {
  1: i32 num1 = 0,
  2: i32 num2,
  3: Operation op,
  4: optional string comment,
}

exception InvalidOperation {
  1: i32 whatOp,
  2: string why
}

service Calculator extends shared.SharedService {
  void ping(),
  i32 add(1:i32 num1, 2:i32 num2),
  i32 calculate(1:i32 logid, 2:Work w) throws
       (1:InvalidOperation ouch),
  oneway void zip()
}
```

当前，读者不必担心该文件中的内容，但需要意识到 RPC Thrift 组合所提供的灵活性，并注意下列一行代码：

```
service Calculator extends shared.SharedService { ...
```

Thrift 支持模板文件间的继承机制，并可供代码生成器使用。

当利用 Thrift 创建客户机/服务器时，可简单地使用下列命令行：

```
$ thrift -r --gen py file_name.thrift
```

上述代码行将在 Python 编程语言中创建客户机和服务器。

在上述众多选择方案中，当前较为流行的是 Thrift 和 gRPC。对于微服务间的直接通信来说，它们都是较好的部署工具。

2.3.2　直接通信间的警示信息

微服务之间的直接通信可导致 DeathStar 问题。DeathStar 是一种反模式，且在递归微服务之间进行通信，从而使得处理过程变得极为复杂，抑或导致产品的开销十分高昂。

利用之前讨论的各种通信工具，可在微服务间构建低延迟的会话。如果未包含处理特定任务的信息，常见的反模式一般支持微服务间彼此自由地交换信息。

此处将会产生警示信息。如果某个微服务总是需要与另一个微服务进行通信，进而完成某项任务，这将产生高耦合信号，并导致 DDD 处理失败，同时还会产生 DeathStar 问题。为了清晰起见，考察下面的示例。

假设存在 4 个微服务，分别表示为 A、B、C、D。其中，某个请求要求提供与 A 相关的信息，但并未包含全部信息内容。对应内容位于 B 和 C 中；但 C 也未包含所有信息，因而请求 D。相应地，D 无法完成分配于它的任务，并从 C 中请求数据。然而，D 需要使用到 A 中的数据。图 2.2 显示了该处理过程的示意图。

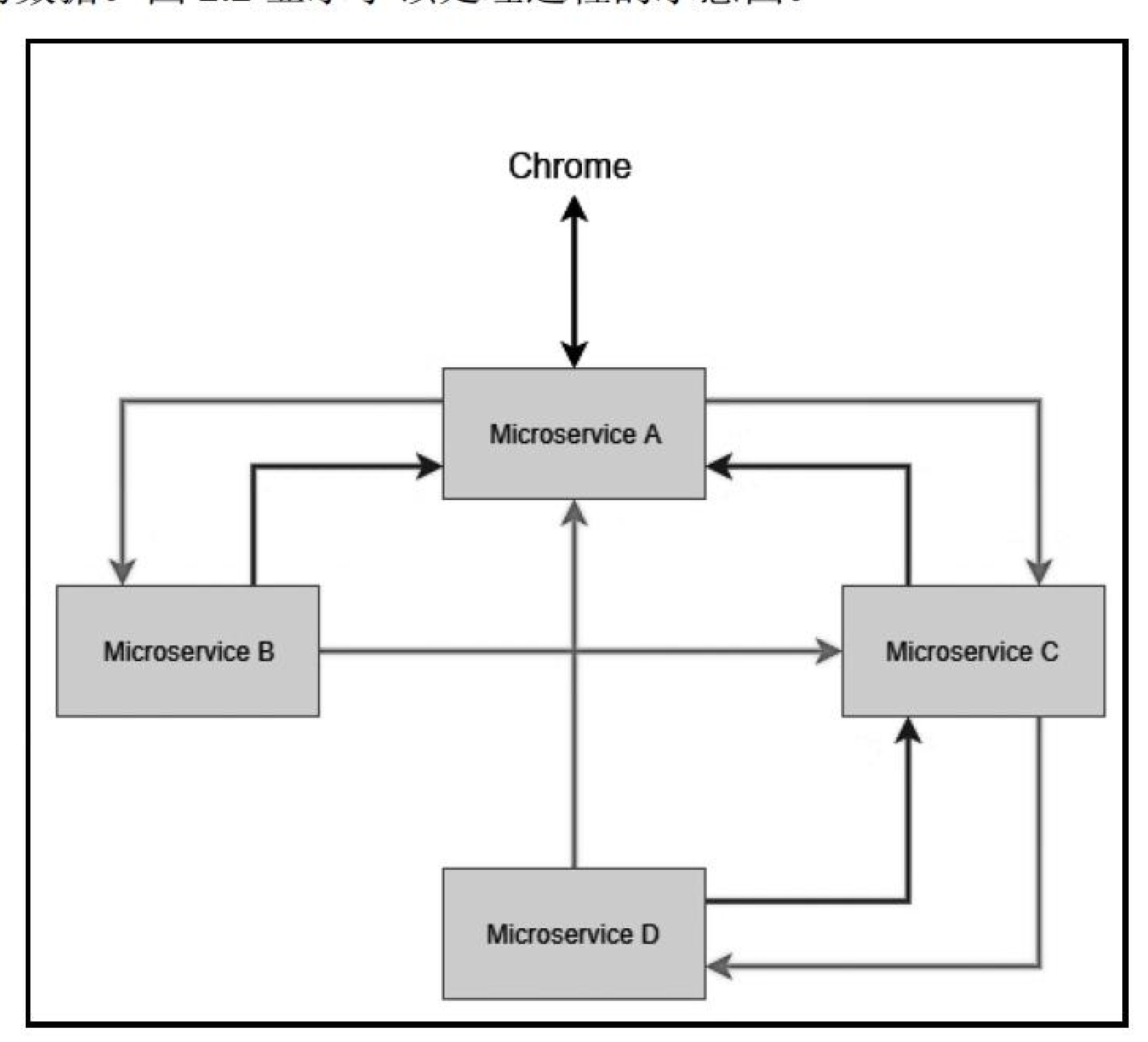


图 2.2

最终，简单的请求将产生复杂的流程，且难以对问题进行监视。虽然这看起来较为自然，但随着时间的推移和新微服务的出现，当前生态系统将变得不可持续。

对于当前消息类型，微服务需要在各自的职责范围内予以充分定义，以实现最小化操作。

无论通信和序列化信息的速度有多快，如果产品缺少人性化特征且难以理解，那么，将很难维持微服务的生态系统，尤其是错误控制方面。

2.4 消息代理——服务间的异步通信

前述内容讨论了基于二进制和 REST 替代方案的微服务之间的同步通信。本节将采用消息代理处理微服务间的通信。也就是说，包含物理内容、通信层以及消息总线的消息系统。

利用消息系统，将不会再产生 DeathStar 问题。在更为健壮的应用程序中，DeathStar 的设计方式如图 2.3 所示。

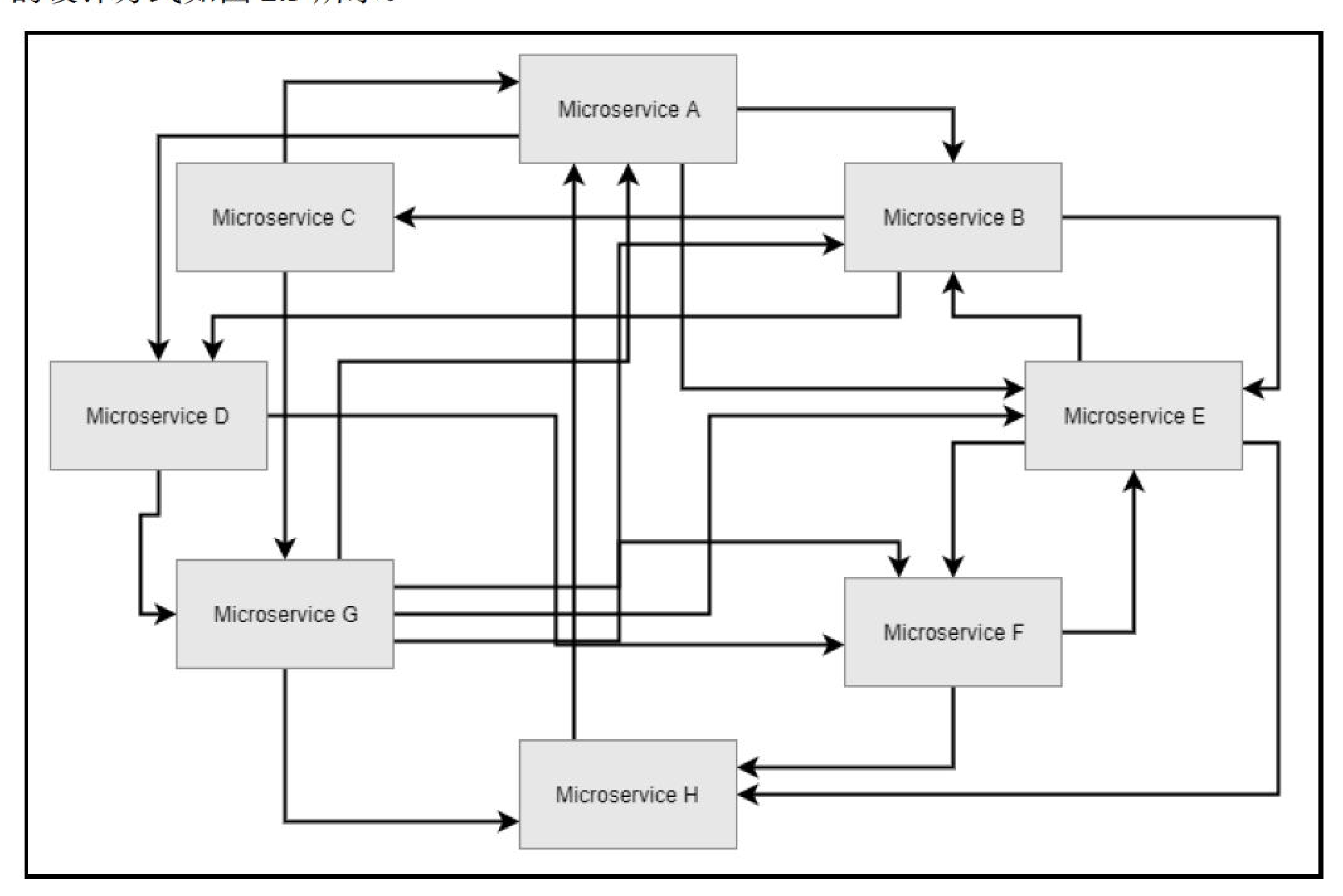


图 2.3

其中，消息系统则截然不同，并与图 2.4 显示的设计方式类似。

消息总线可以用于同步和异步通信中，但需要强调的是，消息总线位于异步通信中。

这里，读者可能会感到疑惑，如果消息图更简单，并且可以使用这种工具进行同步通信，为什么不将这种消息传递机制用于微服务之间所有类型的通信中呢?

这一问题的答案很简单。消息总线是微服务栈中的物理组件，需要像其他基于数据

存储和缓存的物理组件一样进行扩展。这意味着，当使用大容量消息时，同步通信模式可能会导致进程响应出现不必要的延迟。

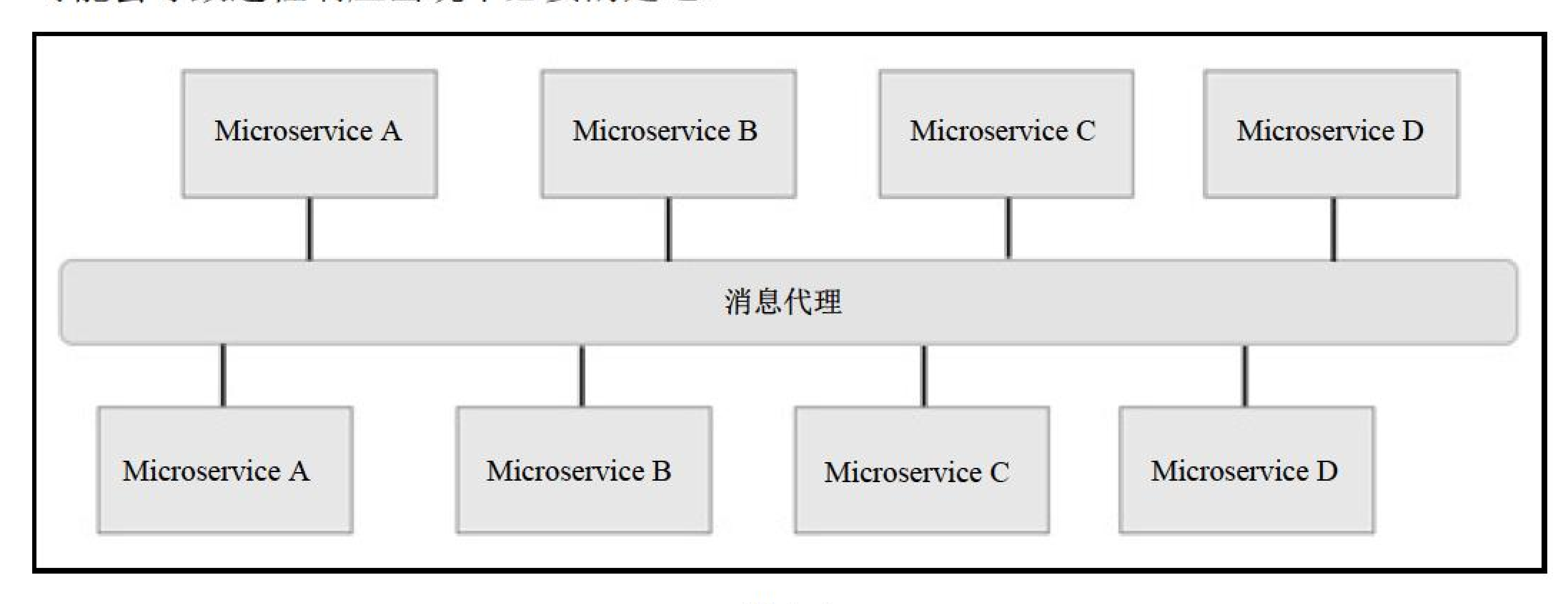


图 2.4

对于工程团队来说，了解如何正确地使用每种工具，且不会对堆栈产生负面影响是非常重要的。

在各种消息代理中，一些方案表现得更加突出，例如：

- ActiveMQ。
- RabbitMQ。
- Kafka。

下面对其进行逐一讨论。

2.4.1 ActiveMQ

ActiveMQ 历史悠久且应用广泛。多年来，它就像是 Java 中的标准消息总线。ActiveMQ 的成熟度和健壮性是毋庸置疑的。

ActiveMQ 支持市场上所使用的大多数编程语言。另外，ActiveMQ 中的问题大多与最常见的通信协议 STOMP 有关。相应地，大多数成熟的库均使用了 ActiveMQ STOMP，但它并不是发送多消息执行者的模型（之一）。ActiveMQ 一直在为 OpenWire 开发一个替代 STOMP 的解决方案，但到目前为止，它只适用于 Java、C、C++和 C#。

ActiveMQ 易于实现且一直处于不断演变中，同时具有良好的文档。如果我们的新闻门户网站应用程序是在 Java 平台或任何支持 OpenWire 的其他平台上开发的，那么 ActiveMQ 是需要仔细考虑的一个方案。

2.4.2 RabbitMQ

在默认情况下，RabbitMQ 使用 AMQP 作为通信协议，这使得它成为传递消息的实用工具。此外，RabbitMQ 文档还对最常用的语言提供了本地支持。

在众多消息代理中，RabbitMQ 因其实用性、与其他工具结合的灵活性，以及简单直观的 API 而表现得十分突出。

下列代码显示了使用 RabbitMQ 在 Python 中创建 Hello World 系统：

```
# import the tool communicate with RabbitMQ
    import pika
    # create the connection
    connection = pika.BlockingConnection(
      pika.ConnectionParameters(host='localhost'))
    # get a channel from RabbitMQ
    channel = connection.channel()
    # declare the queue
    channel.queue declare(queue='hello')
    # publish the message
    channel.basic publish(exchange='',
                      routing key='hello',
                      body='Hello World!')
    print(" [x] Sent 'Hello World!'")
    # close the connection
    connection.close()
```

在上述示例中，我们使用了官方的 RabbitMQ 站点，并针对 hello 发送 Hello World 消息队列。下面是获取消息的代码：

```
# import the tool communicate with RabbitMQ
import pika
# create the connection
connection = pika.BlockingConnection(
pika.ConnectionParameters(host='localhost'))

# get a channel from RabbitMQ
channel = connection.channel()

# declare the queue
channel.queue declare(queue='hello')
# create a method where we'll work with the message received
def callback(ch, method, properties, body):
```

```
  print(" [x] Received %r" % body)

# consume the message from the queue passing to the method
      created above
channel.basic consume(callback,
                      queue='hello',
                      no ack=True)

print(' [*] Waiting for messages. To exit press CTRL+C')

# keep alive the consumer until interrupt the process
channel.start_consuming()
```

上述代码较为简单且具有可读性。RabbitMQ 的另一个特性是扩展使用工具。一些性能测试表明，RabbitMQ 每秒可支持 20000 条消息（每个节点）。

2.4.3　Kafka

Kafka 并不是市场上最简单的消息代理，尤其是在内部工作原理方面。但它是目前为止最具扩展性的，且是一类适用于传递消息的消息代理。

与 ActiveMQ 和 RabbitMQ 不同的是，在过去一段时间中， Kafka 并不发送事务性消息，其原因在于，在应用程序中丢失信息的成本较高。但是在 Kafka 的最新版本中，这一问题已得到解决，而且目前还包含了事务发布选项。

在 Kafka 中，一些基准测试表明，Kafka 可以轻松地支持超过 100000 条消息（每个节点、每秒）。

关于 Kafka，另一个需要重点关注的问题是与其他工具间的集成，如 Apache Spark。另外，Kafka 包含了丰富的文档内容。然而，Kafka 对于编程语言的支持尚有待提高。

对于性能要求较高的场合，Kafka 是一类较好的选择方案。

注意：

对于当前的新闻门户网站，考虑到工具的兼容性和性能，我们选用了 RabbitMQ，旨在与当前应用程序实现较好的兼容。

2.5　缓 存 工 具

需要注意的是，缓存机制并不是唯一可免除数据库的工具，这是一个需要从战略角

度思考的问题。如果不采取缓存，局限性可随之降低，进而提升应用程序的可执行性。但是，正确选择和设置缓存层对于项目的成功至关重要。

缓存策略涉及使用缓存作为数据库的加载点，下面考察图 2.5。

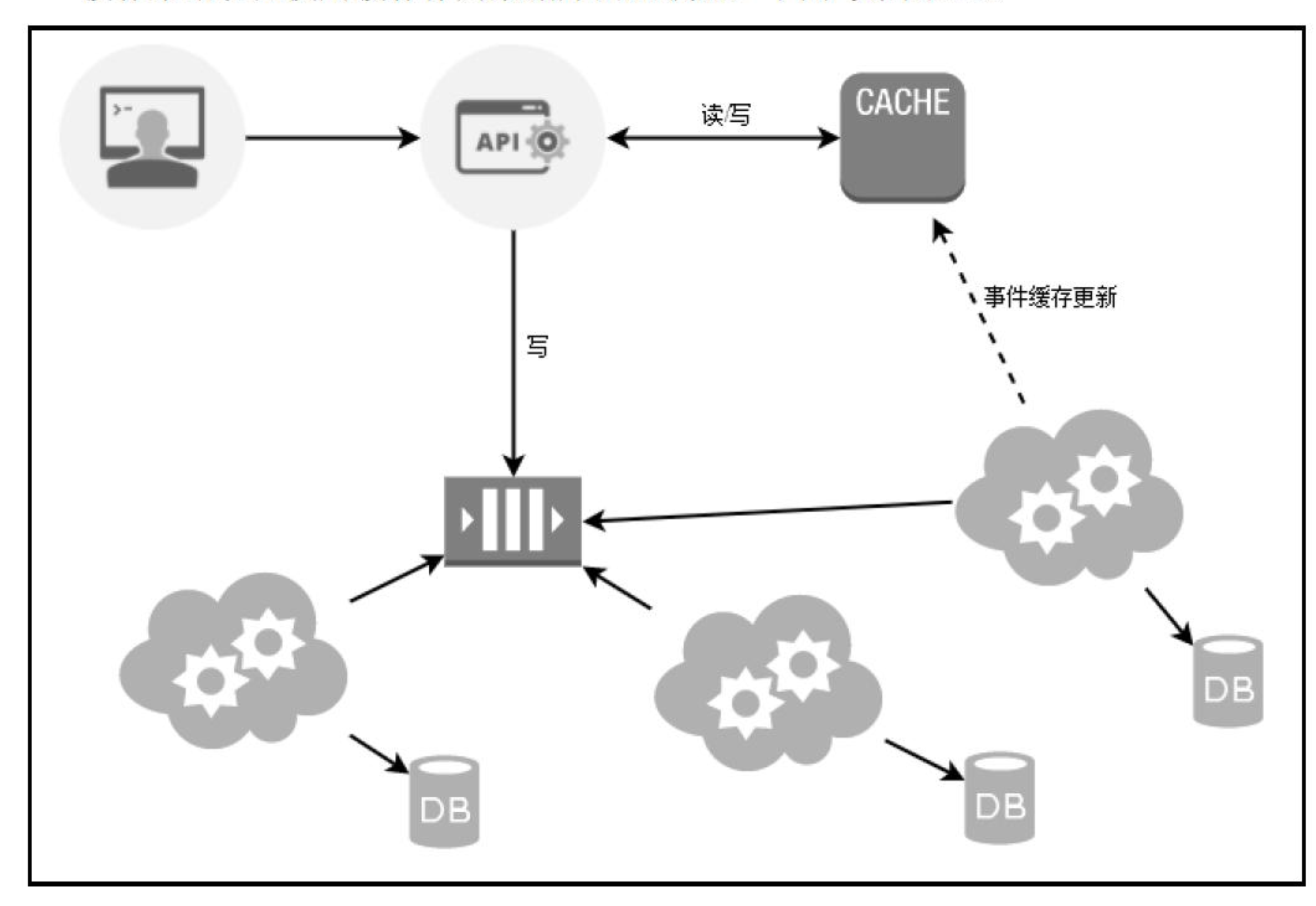


图 2.5

从图 2.5 中可以看到，请求到达 API，但并未直接对其进行处理并发送至数据库中。全部有效请求均被缓存，并被同步置入一行中。

使用者读取队列并处理信息。只有在处理完信息之后，数据才会存储在数据库中。最终，它将在缓存中被重写，以便在数据库中整合数据更新。当使用这一策略时，API 请求的任何信息都将在通过数据库之前直接置于缓存中，以使数据库具备一定的处理时间。

对于终端用户，200 表示为 HTTP 响应。当数据被存储到缓存中时，在数据库中的信息被注册之后，或者当这一过程以异步方式发生时，HTTP 响应均会被发送。

当采用上述策略时，须对现有的工具进行分析。目前，市场上较为著名的工具包括：

- Memcached。
- Redis。

下面对此逐一考察。

2.5.1　Memcached

当谈到 Memcached 时，缓存机制是其最知名、最成熟的应用之一。对于高效的内存应用来说，Memcached 采用了键/值存储这一形式。

对于使用缓存的经典处理过程来说，Memcached 具有简单、实用等特征。另外，Memcached 的性能与内存的使用密切相关。如果 Memcached 使用磁盘注册数据，性能将会严重受损；此外，Memcached 不包含任何磁盘容量的记录，而且始终依赖于第三方工具。

2.5.2　Redis

当涉及缓存时，Redis 实际上可以被看作是市场中的一个新标准。Redis 采用了高效的数据库键/值方案。鉴于其优异的性能，Redis 一直被视为是一种高效的缓存工具。

Redis 文档内容十分丰富且易于理解，即使一个简单的概念也涵盖了许多特性，如 pub/sub 和队列。

由于它的便捷性、灵活性和内部工作模型，Redis 实际上已经将所有其他缓存系统降级为遗留项目。

Redis 内存应用的控制机制非常强大。大多数缓存系统均可有效地从内存中写入和读取数据，但不会清除数据并返回所用的内存。Redis 在这一方面表现得更加优秀，并在清除数据后返回内存以供使用，同时保持了良好的性能。

与 Memcached 不同，Redis 具有本地和可配置的持久化方案。Redis 包含了两种类型的存储形式，即 RDB 和 AOF。

RDB 模型通过快照实现了数据的持久化行为。这意味着，在可配置阶段，内存中的信息将持久化到磁盘上，如下所示：

```
save 60 1000
stop-writes-on-bgsave-error no
rdbcompression yes
dbfilename dump.rdb
```

其中，设置过程较为简单和直观。首先，需要对配置自身进行存储，如下所示：

```
save 60 1000
```

上述代码行表明，如果至少更改了 1000 个键，Redis 应该获取快照，以将数据保存 60 秒。考察下列代码：

```
save 900 1
```

也就是说，如果至少修改了一个键，Redis 每隔 15 分钟对快照执行持久化操作。

当前配置中的第二行代码如下所示：

```
stop-writes-on-bgsave-error no
```

上述代码告知 Redis，即使出现错误，也要继续执行处理和持久化尝试行为。该设置的默认值是 yes，但是如果开发团队决定监视 Redis 的持久化行为，那么，no 则是较好的选项。

通常，Redis 将对数据进行压缩，以节省磁盘空间，对应设置如下所示：

```
rdbcompression yes
```

对于缓存来讲，如果性能问题十分关键，那么，上述值应修改为 no。但是，Redis 所使用的磁盘容量将大幅上升。下列代码设置了 Redis 持久化操作的文件名。

```
dbfilename dump.rdb
```

该名称为配置文件中的默认名称，在不涉及重大问题时，用户可对此进行修改。

其他模型还包括 AOF 的持久化机制。该模型在维持记录数据方面表现得更加安全。然而，相比之下，Redis 的性价比则更高。考察 AOF 配置模板中的下列代码：

```
appendonly no
appendfsync everysec
```

示例中的第一行代码显示了 appendonly 命令，该命令表示是否必须激活 AOF 持久性模式。

在示例配置的第二行 diamante 中，我们可以看到：

```
appendfsync everysec
```

appendfsync 策略通知操作系统，在最快的磁盘上执行持久化操作，而不是缓冲区。appendfsync 包含了 3 种配置模式：no、everysec 和 always，具体解释如下。

- no：禁用 appendfsync。
- everysec：表明应快速执行数据存储；通常情况下，该处理过程一般会延迟 1 秒。
- always：表示更快的持久化处理，最好是立即执行。

读者可能会产生疑问，为何如此关注 Redis 的持久化操作？其原因在于，我们必须确切地了解持久化缓存中的相关功能及其应用方式。

某些开发团队也使用 Redis 作为消息代理。该工具虽然速度较快，但并不适用于该任务——消息传递中缺少了事务。考虑到数量之多，微服务之间的消息可能会丢失。因此，Redis 的适用场合依然是缓存。

2.6　故障警示工具

在享受成功的同时，我们也必须为失败做好准备。在微服务中没有什么比静默错误更加糟糕。因此，尽快接收错误警示信息是至关重要的，这也是一个健康的微服务生态系统应有的标志。

微服务至少包含 4 个主要的故障类型。如果解决了这一类问题，可以说大约 70%的应用程序是安全的。相关场景如下所示：

- ❑ 性能。
- ❑ 构建。
- ❑ 组件。
- ❑ 实现过程中的故障。

下面将对故障点加以逐一讨论，以及如何尽快接收故障警示信息。

2.6.1　性能

这里将进一步研究一些实用工具来证明端点的性能。本地端点测试有助于对性能问题进行预测，这些问题一般仅会在产品中看到。

在将微服务发送到生产环境之后，可以通过一些工具监视整体性能，如 New Relic 和 Datadog。两者均易于实现且包含了丰富的信息显示功能，如图 2.6 所示。



图 2.6

图 2.7 显示了 DATADOG 界面。

图 2.7

性能监视工具中包含了一些免费的选取方案，例如包含 Grafana 和 Prometheus 的 Graphite。为了获取类似的效果，这一类免费方案需要提供更多的设置。

在众多免费的解决方案中，Prometheus 值得一提，其中包含了丰富的信息和实现功能。除了 Graphite 之外，Prometheus 还可与 Grafana 集成，进而显示图形化的性能信息。图 2.8 展示了 Prometheus 界面显示效果。

2.6.2　构建

构建过程是较为重要的一步，同时也是定位故障、且不会对终端用户产生影响的最后一步。微服务架构中的核心内容之一便是自动化处理问题，构建和部署也不例外。

构建的时间点通常是将应用程序转移到特定环境、质量环境或阶段生产之前的最后一个阶段。

在微服务中，所有的功能都必须经历单元测试、功能测试和集成测试。然而，许多开发团队并没有对自动化测试投入过多的注意力，并因此而遭受损失。

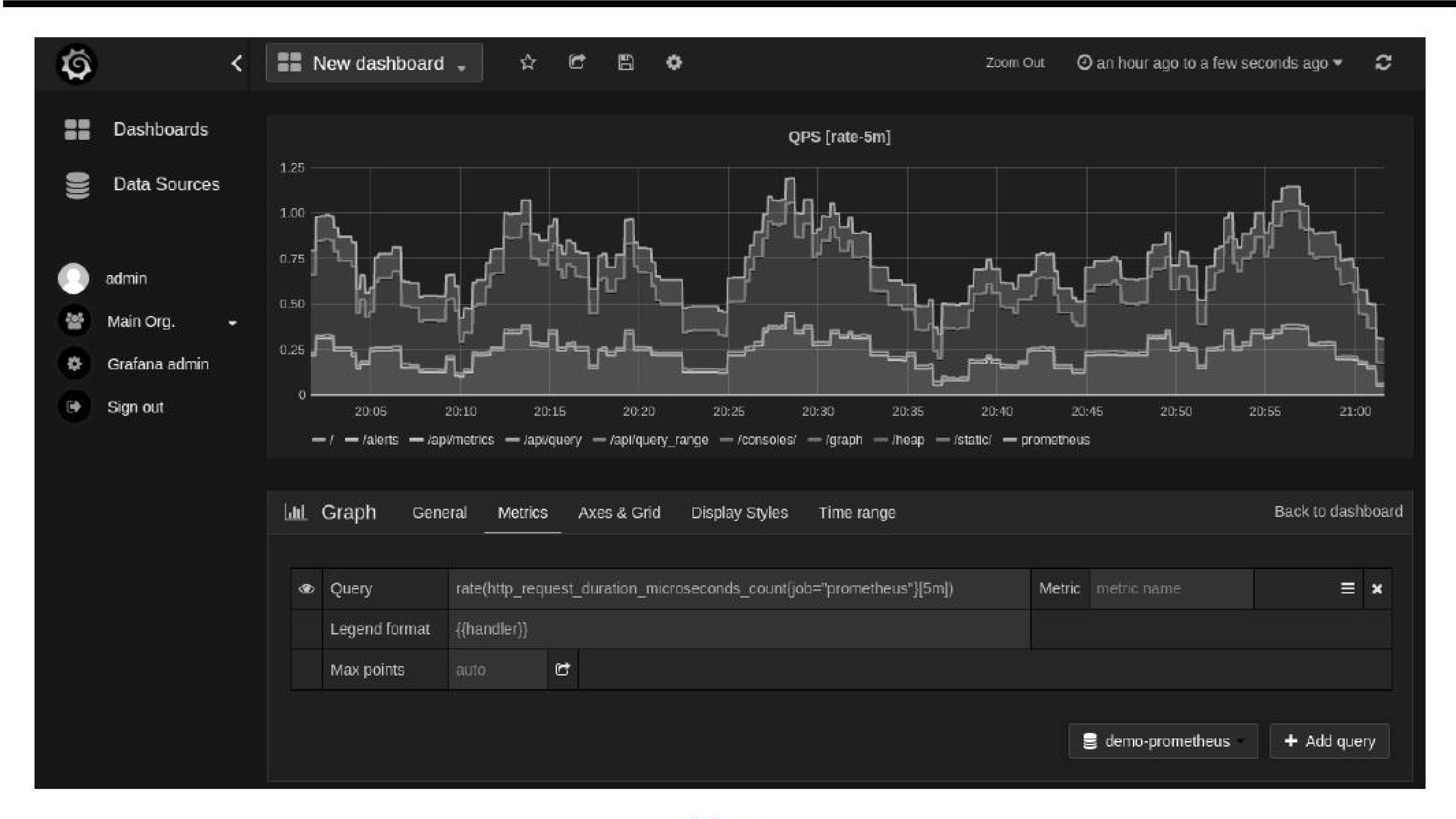


图 2.8

为了实现应用程序构建的自动化过程，从根本上来讲，应用程序部署是一个很好的持续集成工具（或 CI）。在这方面，最为成熟、完整、有效的工具之一即是 Jenkins。Jenkins 是一个免费的开源项目并可进行配置，从而能够完整地实现自动化流程。

除此之外，还存在一些其他解决方案，如 Travis。Travis 可通过在线方式与 CI 协同工作，同时也是一个完全免费的开源项目。Travis 的特点在于可与 GitHub 实现较好的兼容。

使用 CI 的最重要的因素是，可正确地设置应用程序测试过程，如前所述，这是在影响产品最终用户之前捕获故障的最后阶段。CI 是集成微服务测试的最佳位置。

2.6.3　组件

微服务架构的一个显著特点是，大量的组件可能会出现故障。相应地，容器、数据库、缓存以及消息代理均可视作故障点的例子。

假设数据库的硬盘在某些物理组件中出现了故障，从而导致应用程序无法正常运行。如果未对此类问题进行监视，那么，围绕应用程序的处理时间通常很高——应用程序、开发和支持团队都会对软件故障展开调查。只有在确认了故障不在软件中之后，团队才能继续在物理组件中查找问题。

一些工具，诸如 pens-sentinel，可提供更大的灵活性，但并非所有物理组件均包含这一类支持。

一种简单的解决方案是在每个微服务中创建一个检查端点，该端点不仅负责验证微服务实例（无论它是否处于运行状态），同时还负责验证微服务连接到的所有组件。

2.6.4　实现鸿沟

在某些情况下，自动化测试可能存在缺陷，且未能考虑到所有的情况，或者一些外部组件（如 API 供应商）会在应用程序中抛出错误。

一般情况下，此类错误均为静默错误，仅在用户报告该错误之后方可被发现。但这里的问题是，有多少用户经历了此类错误，但并未提交相关报告？产品中是否包含了价值损失级别方面的错误？

此类问题并不存在明确的答案，且几乎难以实现量化操作。为了快速捕捉这一类问题，我们需要对应用程序的内部故障进行监视。

针对这一类监视类型，存在大量的工具，其中 Sentry 表现得较为突出。Sentry 包含以下特征：

- ❑ 实时查看最新部署所产生的影响。
- ❑ 针对被错误中断的特定用户，向其提供相关支持。
- ❑ 检测并消除各种欺骗行为：购买、身份验证和其他关键领域中出现异常数量的故障。
- ❑ 外部集成。

需要说明的是，Sentry 的免费版本无法提供功能全面的解决方案。

如果警示系统完美地覆盖了上述 4 个故障点，那么，我们就可以安全投入到开发过程中，并通过自动化和持续处理将微服务置入到生产中去。

2.7　数　据　库

当采用语言和编程框架进行程序设计时，当前数据库方案尚无法涉及全部应用场合。

选择使用数据库对于评估各项功能和微服务操作模式是十分必要的。在某些情况下，关系数据库仍具有实际意义。有时，NoSQL 可能是更好的解决方案。当然，在一些场合下，这些数据库中可能均无法满足要求。例如，我们需要通过图形方式使用数据库，这也是当前新闻门户网站所面临的情况。

实际上，SQL 尚无法满足全部功能。对于当前各项微服务，我们需要选取最为适宜的数据库类型。对此，下列应用程序会涉及这一问题：

- ❑ SportNewsService。
- ❑ PoliticsNewsService。
- ❑ FamousNewsService。
- ❑ RecommendationService。
- ❑ UsersService。

对于直接显示新闻的微服务，如果仔细考虑这一问题，会发现其间存在一个直接的关系系统。此类微服务的目标是简单而快速地批量显示新闻主题。这种行为非常接近 NoSQL 的功能——在 NoSQL 中，关系结构较弱，而搜索等基本操作的速度较快。

对于之前提及的各种服务和特性，我们采用了 NoSQL 类型的数据库解决方案，即 MongDB。MongDB 包含了丰富的文档，且兼具实现简单和低成本等特征。

UsersService 则完全不同。应用程序用户证书包含登录和密码输入操作，其中会涉及与此相关的其他数据，例如注册数据和默认的偏好设置内容。因此，这一领域包含了关系型信息，因此传统的 SQL 较为适宜。

UsersService 可使用 MariaDB、MySQL、Oracle、SQL Server 或 PostgreSQL。在当前示例中，考虑到兼容性、成熟度，以及与堆栈兼容关系，我们使用了 PostgreSQL。

RecommendationService 所涉及的各项功能与其他微服务截然不同，且至少需要在偏好设置和用户之间构建对应关系。除此之外，还应提供同一主题下用户的关注方式和数量。对此，可利用传统的 SQL 数据库创建这种关系类型。然而，随着时间的推移，将会出现复杂的查询操作；对于速度、维护方面的错误理解，概念堆栈中的错误选择方案将对微服务带来负面影响。

针对 RecommendationService，数据库 Neo4J 可视作一种较好的方案。在当前微服务中，图形的质量以及工具的简单性正是我们所追求的目标。

当涉及数据库时，主要目标是理解字段的行为方式，同时避免将其置于“舒适区域”（comfort zone）中。重要的是，需要针对各种场合选取最为适宜的工具。

2.8　本地性能度量

当与微服务架构协同工作时，最糟糕的情形之一即是在产品中添加代码，从而导致性能大幅下降。当代码返回至开发环境时，将会发现项目产品受损严重，用户经历了糟糕的体验，而这一切却可通过技术手段予以分析，这种感觉的确令人沮丧。

当前，可对产品中的问题进行预测，甚至可在开发环境中加以解决。当注册这种类型的度量方式时，存在大量的工具可在本地环境中对性能问题进行验证。

显然，本地行为并不能很好地反映生产环境，还需要对诸多因素加以考虑，如网络延迟、用于部署和生产的机器设备，以及与外部工具间的通信。然而，我们可通过本地度量方法以彰显影响应用程序整体性能的最新算法以及相关功能。

对于本地应用程序度量方法，包含以下一些工具：

- Apache Benchmark。
- WRK。
- Locust。

其中，每种工具均包含各自的特性，但它们的目的都是相同的，即获取端点的度量结果。

2.8.1 Apache Benchmark

Apache Benchmark 简称为 AB，后续章节将采用这一称谓。

AB 可通过命令行方式运行，对于验证端点的速度和响应来说十分有效。

运行局部性能测试十分简单，如下所示：

```
$ ab -c 100 -n 10000 http://localhost:5000/
```

上述命令行将调用 AB（-n 10000）、模拟 100 个并发用户（-c 100），并调用本地端口 5000（http:// localhost:5000/）。对应显示结果如图 2.9 所示。

```
Server Software:        Werkzeug/0.12.2
Server Hostname:        localhost
Server Port:            5000

Document Path:          /
Document Length:        11 bytes

Concurrency Level:      100
Time taken for tests:   2.247 seconds
Complete requests:      1000
Failed requests:        0
Total transferred:      165000 bytes
HTML transferred:       11000 bytes
Requests per second:    444.99 [#/sec] (mean)
Time per request:       224.722 [ms] (mean)
Time per request:       2.247 [ms] (mean, across all concurrent requests)
Transfer rate:          71.70 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   0.1      0       1
Processing:    59  210  37.5    221     231
Waiting:       53  210  38.2    221     231
Total:         59  210  37.5    221     232
```

图 2.9

上述结果显示了执行处理的服务器（Werzeug/0.12.2）、主机名（localhost）、端口（5000）和另一组信息。

AB 所生成的最为重要的数据包括以下内容。

- 每秒的请求数量：444.49。
- 每个请求的时间：224.722 毫秒（平均值）。
- 每个请求的时间：2.247 毫秒（所有并发用户的平均值）。

测试结束时，上述 3 个信息体现了本地应用程序的性能。不难发现，对于 100 个并发用户和 10000 个请求，当前测试中使用的应用程序每秒返回 444.99 个请求。

显然，这是使用 AB 完成的最为简单的测试场景，该工具还有许多其他特性，例如导出图形性能测试结果，以及模拟 REST API 在 HTTPS 证书上运行和模拟的所有谓词（verb）。需要说明的是，这些只是 AB 作为资源提供的其他一些属性。

2.8.2 WRK

类似于 AB，WRK 也是一类命令行执行工具，并具备与 AB 相同的功能。图 2.10 显示了 WRK 工具。

wrk - a HTTP benchmarking tool

wrk is a modern HTTP benchmarking tool capable of generating significant load when run on a single multi-core CPU. It combines a multithreaded design with scalable event notification systems such as epoll and kqueue.

An optional LuaJIT script can perform HTTP request generation, response processing, and custom reporting. Details are available in SCRIPTING and several examples are located in scripts/.

Basic Usage

```
wrk -t12 -c400 -d30s http://127.0.0.1:8080/index.html
```

This runs a benchmark for 30 seconds, using 12 threads, and keeping 400 HTTP connections open.

Output:

```
Running 30s test @ http://127.0.0.1:8080/index.html
  12 threads and 400 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency   635.91us    0.89ms  12.92ms   93.69%
    Req/Sec    56.20k     8.07k  62.00k    86.54%
  22464657 requests in 30.00s, 17.76GB read
Requests/sec: 748868.53
Transfer/sec:    606.33MB
```

图 2.10

运行 WRK 十分简单，如下列命令行所示：

```
$ wrk -c 100 -d 10 -t 4 http://localhost:5000/
```

然而，与 AB 相比，WRK 也包含了某些不同的特性。上述命令表示，WRK 执行了 10 秒的性能测试（d 10），其中涉及 100 个并发用户，并针对当前任务从操作系统中请求 4 个线程（-t 4）。

上述命令行并未执行限制或请求加载语句。WRK 并未采取这种方式工作，而是在一段时间内执行负载应力测试。

10 秒后，WRK 将显示如图 2.11 所示的信息。

```
Running 10s test @ http://localhost:5000/
  4 threads and 100 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency   268.68ms   73.98ms 678.53ms   89.14%
    Req/Sec    92.41     25.42   141.00     69.17%
  3688 requests in 10.09s, 594.26KB read
Requests/sec:    365.55
Transfer/sec:     58.90KB
```

图 2.11

显然，返回的数据更为简洁。针对当前应用程序，读者仅需了解临时变更前的一些操作行为即可。

再次强调，较好的方法是显示本地测试特性；对于产品中应用程序的真实状态，WRK 的结果不一定能够作为最终证据。尽管如此，WRK 依然可提供较好的数据结果，进而反映应用程序的度量结果。

根据 WRK 生成的数据，不难发现，在使用 100 个并发用户以及 4 个线程进行 10 秒测试之后，本地环境中的应用程序将显示以下数字。

- 请求/秒：365.55。
- 268.68 毫秒的延迟（平均值）。

WRK 数据略低于 AB 提供的数据；显然，每种工具所执行的测试结果稍有不同。

在运行测试方面，WRK 表现得十分灵活。例如，可使用 Lua 编程语言中的脚本，并执行特定任务。

WRK 是作者最喜欢的本地性能测试工具之一。WRK 所执行的测试类型与现实情况非常接近，所提供的数字也非常接近于实际结果。

2.8.3 Locust

在本地度量 API 所列出的工具中，Locust 配置了一个可视化界面。另一个值得关注

的特性则是，Locust 可同时验证多个端点。

Locust 界面简单且易于理解，读者可在界面数据输入中查看到所用的并发用户数量。在利用 Locust 开启处理过程后，首先将显示所用的 GUI HTTP 谓词、请求被指向的路径、测试期间发出的请求数量，以及从多个站点采集的度量结果的编号。

Locust GUI 的详细信息如图 2.12 所示。

LOCUST
A MODERN LOAD TESTING TOOL
STATUS RUNNING 9600 users Edit
SLAVES 6
RPS 76.4
FAILURES 0%
STOP
Reset Stats

Statistics Failures Exceptions

Type	Name	# requests	# fails	Median	Average	Min	Max	Content Size	# reqs/sec
GET	/	1831	0	21	21	4	38	19947	18.3
GET	/blog	608	0	25	26	3	49	19841	6.9
GET	/blog/[post-slug]	612	0	14	15	2	27	19858	7.8
GET	/forum	573	0	26	26	3	49	20209	5.5
GET	/forum/[thread-slug]	596	0	30	30	6	55	20209	5.3
POST	/forum/[thread-slug]	71	0	62	63	13	120	11188	0.6
POST	/forum/new	64	0	59	58	6	108	3272	0.7
GET	/signin	3439	0	26	26	3	49	19850	31.3
	Total	7794	0	26	25	2	120	19711	76.4

图 2.12

Locust 使用起来十分简单。第一步是安装过程。与 AB 和 WRK 不同，Locust 的安装过程通过 pypi 完成，即 Python 安装包，并输入下列命令：

```
$ pip install locustio
```

在安装完毕后，须创建一个名为 locustfile.py 的配置文件，并包含以下内容：

```
# import all necessary modules to run the Locust:
 from locust import HttpLocust, TaskSet, task

    # create a class with TaskSet as inheritance
    class WebsiteTasks(TaskSet):
    # Create all the tasks that will be metrify using the @taks decorator
    @task
    # the name function will be the endpoint name in the Locust
    def index(self):
        # set to the client the application path with the HTTP
```

```
        verb # in this case "get"
      self.client.get("/")
      @task
      def about(self):
        self.client.get("/about/")

   # create a class setting the main task end the time wait for each #
request
     class WebsiteUser(HttpLocust):
      task set = WebsiteTasks
      min wait = 5000
      max_wait = 15000
```

在文件配置完毕后，即可运行 Locust。对此，可输入下列命令行：

```
$ locust -f locustfile.py
```

Locust 将提供一个 URL 进而访问可视化界面，并于随后对度量结果进行验证。

在开始阶段，与前述应用程序相比，Locust 配置过程可能稍显复杂。但随着过程的不断深入，测试过程将变得十分简单。正如 AB 和 WRK 那样，Locust 包含许多特性可以进行更深入的测试。

2.9 本章小结

本章讨论了选择微服务堆栈的重要性。开始时，制定相关决策似乎很复杂，但如果进一步考察开发领域的定义，这一过程将变得相对简单。

我们已经看到，编程语言、框架和数据库实现了目标的定义。一个简单的例子是，火车无法实现飞行任务。也就是说，某些工具并不适用于特定目的。

另外，本章还讨论了缓存的重要性、微服务间快速和敏捷通信的构建方式，以及微服务多个层中故障警报的重要性。

最后，本章还介绍了一些工具，这些工具可以帮助我们验证微服务在本地环境中的性能。

根据本章所介绍的知识，读者可顺利地进入下一章的学习，并通过较为高效的方式创建微服务。

在第 3 章中，将对第一个微服务展开编码工作。

第 3 章　内 部 模 式

前两章讨论了领域驱动设计（DDD）、缓存机制和数据库，这些内容对于开发高效和可伸缩的微服务至关重要。本章将涉及更多的理论知识，尤其是基于微服务开发的设计模式。

下面将介绍针对微服务的模式、反模式、工具以及代码结构。本章将围绕缓存、队列、异步机制以及 worker 的应用展开讨论，并着手编写应用程序，进而将理论付诸于实践。在阅读完本章后，读者将了解两种有效的微服务模式，即 CQRS 和事件源。

本章主要涉及以下内容：

- 开发结构。
- 缓存策略。
- CQRS——队列策略。
- 事件源——数据集成。

3.1　开 发 结 构

第 2 章曾定义了 5 个域，分别包括：

- SportNewsService。
- PoliticsNewsService。
- FamousNewsService。
- RecomendationService，
- UsersService。

这里，第一步是选取某个域以实施相关技术。对此，我们将使用到 UsersService。该领域仅包含了唯一特征，因而是应用相关技术的绝佳场所。

下面针对 UsersService 组合探讨相关工具。在开始阶段，我们仅使用单一数据库。

3.1.1　数据库

此处将使用 PostgreSQL 数据库，图 3.1 显示了第一个表结构，其内容较为简单。

```
                                  Table "public.users"
  Column  |  Type   |                      Modifiers                      | Storage  | Stats target | Description
----------+---------+-----------------------------------------------------+----------+--------------+-------------
 id       | integer | not null default nextval('users_id_seq'::regclass) | plain    |              |
 name     | text    | not null                                            | extended |              |
 email    | text    | not null                                            | extended |              |
 password | text    | not null                                            | extended |              |
Indexes:
    "users_pkey" PRIMARY KEY, btree (id)
```

图 3.1

表 users 定义了一个 ID 并作为唯一的标识键；除此之外，表中还设置了用户名、用户电子邮件以及用户密码。

3.1.2　编程语言和工具

UsersService 将采用 Go 语言进行编写，第 2 章曾对此有所介绍。这里，还需要下载项目的依赖关系并以此启动当前项目。对此，可输入下列命令：

```
$ go get github.com/gorilla/mux
$ go get github.com/lib/pq
$ go get github.com/codegangsta/negroni
$ go get github.com/jmoiron/sqlx
```

其中，Muxer 负责处理应用程序句柄，同时也是初始状态下的依赖关系。gorilla/mux 负责在开始阶段处理 UsersService 同步通信层，该通信使用了 HTTP/JSON。

作为接口，PQ 负责软件和 PostgreSQL 数据库间的通信，本章将对其工作方式加以讨论。

Negroni 表示为中间件管理器。在应用程序阶段，其唯一职责是管理应用程序日志。

SQLX 表示数据库中队列的执行者。考虑到 SQLX 在 Go 语言中十分常见，因而基本与微服务中 ORM 的使用方式并无太大差别。然而，与当前结构相关的、查询中所返回的数值应用则通过 SQLX 运行。

3.1.3　项目结构

结构的简单性是微服务的主要特征，但这并不意味着健壮性的缺失，而是指项目开发过程易于阅读和理解。

相应地，UsersService 使得后续调整工作相对简单，其间，项目将在其开发周期内被多次修改。

在最初的版本中，项目的结构涵盖了以下内容：

- models.go。

- app.go。
- main.go。

下面对其各项功能进行逐一考察。

1. models.go 文件

模型是理解项目业务规则的绝佳场所。

首先可声明相关工作包，随后是项目所需的导入操作，如下所示：

```
package main

import (
  "github.com/jmoiron/sqlx"
  "golang.org/x/crypto/bcrypt"
)
```

接下来将进行实体 Users 的声明。注意，Go 语言并不包含类，而是使用了结构。虽然最终行为与 OOP 类似，但结构依然包含了自身的某些独特之处，如下所示：

- 结构 User 负责显示数据实体，对应代码如下：

```
type User struct {
  ID        int     `json:"id" db:"id"`
  Name      string  `json:"name" db:"name"`
  Email     string  `json:"email" db:"email"`
  Password  string  `json:"password" db:"password"`
}
```

因此，我们创建 5 个基本的 CRUD 操作，即 five-create、read one、read list、update 和 delete。

- 当获取返回结果时，仅需利用源自 db 的数据更新 User 实例即可，如下所示：

```
func (u *User) get(db *sqlx.DB) error {
  return db.Get(u, "SELECT name, email FROM users WHERE id=$1", u.ID)
}
```

这里，get 方法仅返回一个用户。对于 ID，要搜索的数据将在结构体的内存引用中被传递，这一点在语法 u.ID 中可以看到。需要注意的是，PostgreSQL 中数据库的访问是通过依赖输入完成的，并作为 get 方法中的参数予以传递。

- 利用实例值更新 db 中的数据，如下所示：

```
func (u *User) update(db *sqlx.DB) error {
  hashedPassword, err := bcrypt.GenerateFromPassword(
```

```
        []byte(u.Password),
        bcrypt.DefaultCost,
  )
  if err != nil {
    return err
  }
   , err = db.Exec("UPDATE users SET name=$1, email=$2,
   password=$3 WHERE id=$4", u.Name, u.Email,
   string(hashedPassword), u.ID)
  return err
}
```

update 方法类似于结构中的 get 方法。其中，持久化数值在指针*Users 中传递，数据库访问则基于依赖输入。除了查询操作之外（显然不同于 get 方法中的查询操作），此处还展示了一种密码加密机制。为了确保密码不会在数据库中予以显示，这里采用了 Go 语言中的 bcrypt 库，并根据传递至当前方法中的密码生成哈希值。

❑ 利用实例值删除 db 中的日期，如下所示：

```
func (u *User) delete(db *sqlx.DB) error {
  _, err := db.Exec("DELETE FROM users WHERE id=$1", u.ID)
  return err
}
```

delete 方法与 get 方法包含相同的结构。此处应注意语法“_, err :=db.Exec(...)”，对应的语法结构类型稍显奇特，但却不可或缺。Go 语言并不会抛出异常，只是简单地返回一个错误信息。这种控制类型较为另类，但是，不容否认的是，代码中也包含了许多优雅之处。

❑ 利用实例值在 db 中创建新用户，如下所示：

```
func (u *User) create(db *sqlx.DB) error {
  hashedPassword, err := bcrypt.GenerateFromPassword(
    []byte(u.Password),
    bcrypt.DefaultCost,
  )
  if err != nil {
    return err
  }
  return db.QueryRow(
    "INSERT INTO users(name, email, password) VALUES($1, $2, $3)
     RETURNING id", u.Name, u.Email,
```

```
    string(hashedPassword)).Scan(&u.ID)
}
```

update 方法针对添加至数据库中的密码进行处理，对应代码稍后予以展示。

- List 返回用户列表，并可用于分页操作，如下所示：

```
func list(db *sqlx.DB, start, count int) ([]User, error) {
  users := []User{}
  err := db.Select(&users, "SELECT id, name,
   email FROM users LIMIT $1 OFFSET $2", count, start)
  if err != nil {
    return nil, err
  }
  return users, nil
}
```

最后，我们将讨论模型中最为独特的一点。这一不同之处主要因为列表并不是一个方法，而是定义为一个函数——这在 Go 开发人员中十分常见。只有当更改或使用某个实例或结构时，才会创建相应的方法。另外，创建函数的数据若被更改或加以使用，list 函数将简单地返回一个用户列表（接收信息分页参数）。

归档文件 models.go 包含了如下内容：

```
package main
import (
  "github.com/jmoiron/sqlx"
  "golang.org/x/crypto/bcrypt"
)
```

User 则定义为结构（struct），表示为数据库实体，如下所示：

```
type User struct {
  ID              int        `json:"id" db:"id"`
  Name            string `json:"name" db:"name"`
  Email         string `json:"email" db:"email"`
  Password string `json:"password" db:"password"`
}
```

Get 仅返回基于 db 数据的更新用户实例，如下所示：

```
func (u *User) get(db *sqlx.DB) error {
  return db.Get(u, "SELECT name, email FROM users WHERE id=$1",u.ID)
}
```

下列代码显示了利用实例值更新 db 中的数据。

```
func (u *User) update(db *sqlx.DB) error {
  hashedPassword, err := bcrypt.GenerateFromPassword(
    []byte(u.Password), bcrypt.DefaultCost)
  if err != nil {
    return err
  }
   , err = db.Exec("UPDATE users SET name=$1, email=$2,
   password=$3 WHERE id=$4", u.Name, u.Email,
   string(hashedPassword), u.ID)
  return err
}
```

下列代码显示了利用实例值删除 db 中的数据。

```
func (u *User) delete(db *sqlx.DB) error {
   , err := db.Exec("DELETE FROM users WHERE id=$1", u.ID)
  return err
}
```

下列代码显示了利用实例值创建 db 中的新用户。

```
func (u *User) create(db *sqlx.DB) error {
  hashedPassword, err := bcrypt.GenerateFromPassword(
    []byte(u.Password), bcrypt.DefaultCost)
  if err != nil {
    return err
  }
  return db.QueryRow(
    "INSERT INTO users(name, email, password) VALUES($1, $2, $3)
    RETURNING id", u.Name, u.Email,
    string(hashedPassword)).Scan(&u.ID)
}
```

List 返回用户列表，并可用于分页操作中，如下所示：

```
func list(db *sqlx.DB, start, count int) ([]User, error) {
  users := []User{}
  err := db.Select(&users, "SELECT id, name,
   email FROM users LIMIT $1 OFFSET $2", count, start)
  if err != nil {
    return nil, err
  }
  return users, nil
}
```

2．app.go 文件

app.go 文件负责接收数据，并将其发送至数据存储中。当查看该文件时，对应内容较为复杂且分散，但重要的一点是需要理解所用语言的基本特征。

Go 是一种命令式编程语言，旨在编写清晰、简单的目标代码。Go 语言中较少出现诸如 MVC 这一类模式，一般也不涉及复杂的设置结构。当然，这并不意味着此类模式无法使用。恰恰相反，如有必要，类似于 MVC 这种模式也可用于 Go 语言的文件结构中。

下面考察 app.go 文件。再次强调，数据包的声明以及文件中的导入操作不可或缺。对应代码如下所示：

```
package main
import (
  "database/sql"
  "encoding/json"
  "fmt"
  "log"
  "net/http"
  "strconv"

  "github.com/codegangsta/negroni"
  "github.com/gorilla/mux"
  "github.com/jmoiron/sqlx"
    "github.com/lib/pq"
)
```

这里需要注意“_ "github.com/lib/pq"”这一行代码。其中，“_”表示调用该库中的 init 方法。但是，目前尚未在代码中生成指向该库的直接引用。

与 models.go 文件类似，这里同样声明了一个结构。具体来说，App 结构由两个元素构成，即指向 SQLX 的内存引用，以及指向当前路由的内存引用。

作为结构，App 中包含了应用程序配置值，如下所示：

```
type App struct {
  DB                    *sqlx.DB
  Router            *mux.Router
}
```

我们所讨论的第一个结构方法是 Initialize。在当前示例中，我们已经获得了基于实例化数据库的连接。

Initialize 方法将创建 DB 连接，并筹备全部路由，对应代码如下所示：

```
func (a *App) Initialize(db *sqlx.DB) {
  a.DB = db
  a.Router = mux.NewRouter()
  a.initializeRoutes()
}
```

在连接初始化完毕后，下面在 initializeRoutes 方法中定义路由，对应代码如下所示：

```
func (a *App) initializeRoutes() {
  a.Router.HandleFunc("/users", a.getUsers).Methods("GET")
  a.Router.HandleFunc("/user", a.createUser).Methods("POST")
  a.Router.HandleFunc("/user/{id:[0-9]+}",
   a.getUser).Methods("GET")
  a.Router.HandleFunc("/user/{id:[0-9]+}",
   a.updateUser).Methods("PUT")
  a.Router.HandleFunc("/user/{id:[0-9]+}",
   a.deleteUser).Methods("DELETE")
}
```

路由的定义并不复杂，仅是简单的 CRUD 而已。相应地，下列方法用于初始化微服务。其中，Run 方法激活 Negroni，即中间件控制器，并将路由器传递于其中，同时激活服务器。

此外，Run 方法还将初始化服务器，如下所示：

```
func (a *App) Run(addr string) {
  n := negroni.Classic()
  n.UseHandler(a.Router)
  log.Fatal(http.ListenAndServe(addr, n))
}
```

为了避免重复代码，下面在文件中定义两个辅助函数，即 respondWithError 和 respondWithJSON 函数。其中，第一个函数负责传递至 HTTP 层、在应用程序内生成错误代码，如 404 或 500，以及须予以报告的所有代码。

第二个函数负责创建响应每个请求的 JSON，对应代码如下所示：

```
func respondWithError(w http.ResponseWriter, code int, message
 string) {
  respondWithJSON(w, code, map[string]string{"error": message})
}
func respondWithJSON(w http.ResponseWriter, code int, payload
 interface{}) {
  response,   := json.Marshal(payload)
```

```
  w.Header().Set("Content-Type", "application/json")
  w.WriteHeader(code)
  w.Write(response)
}
```

下面着手定义 CRUD 方法。第一个定义的方法负责获取单一用户，此处将其命名为 getUser。该方法将转换请求中的最后一个 ID，即检测该 ID 是否可用于数据库中的搜索操作。如果未抛出 HTTP 错误，那么，将创建 User 实例，并调用模型中的 get 方法。如果数据库中不存在该用户，抑或出现了任意一类数据库故障，则发送相应的 HTTP 错误代码。最后，如果一切就绪，JSON 将被发送以响应当前请求。对应代码如下所示：

```
func (a *App) getUser(w http.ResponseWriter, r *http.Request) {
  vars := mux.Vars(r)
  id, err := strconv.Atoi(vars["id"])
  if err != nil {
              respondWithError(w, http.StatusBadRequest, "Invalid
   product ID")
              return
  }

  user := User{ID: id}
  if err := user.get(a.DB); err != nil {
    switch err {
      case sql.ErrNoRows:
          respondWithError(w, http.StatusNotFound, "User not found")
      default:
          respondWithError(w, http.StatusInternalServerError,
err.Error())
    }
    return
  }

  respondWithJSON(w, http.StatusOK, user)
}
```

app.go 文件的下一个方法的功能类似于 getUser。然而，我们希望能够同时获得多个用户，因而对应方法称之为 getUsers。该方法接收 count 和 start 作为搜索分页的参数，并规范化错误搜索的可能值。随后，该方法使用 models.go 文件中的 list 方法，并传递数据库连接实例，以及搜索分页值。如果一切工作顺利，将发送 JSON 以响应请求。对应代码如下所示：

```
func (a *App) getUsers(w http.ResponseWriter, r *http.Request) {
  count,   := strconv.Atoi(r.FormValue("count"))
  start,   := strconv.Atoi(r.FormValue("start"))

  if count > 10 || count < 1 {
    count = 10
  }
  if start < 0 {
    start = 0
  }

  users, err := list(a.DB, start, count)
  if err != nil {
    respondWithError(w, http.StatusInternalServerError, err.Error())
    return
  }

  respondWithJSON(w, http.StatusOK, users)
}
```

下面继续讨论数据库中的相关方法。其中，第一个方法是 createUser。在该方法中，我们将 JSON 转换为发送至用户实例的请求体。当前用户实例调用模型的 create 方法并持久化数据。如果未产生任何错误，将不会返回 HTTP 201，表明当前用户被成功地创建。对应代码如下所示：

```
func (a *App) createUser(w http.ResponseWriter, r *http.Request) {
  var user User
  decoder := json.NewDecoder(r.Body)
  if err := decoder.Decode(&user); err != nil {
    respondWithError(w, http.StatusBadRequest, "Invalid request
payload")
    return
  }
  defer r.Body.Close()

  if err := user.create(a.DB); err != nil {
    fmt.Println(err.Error())
      respondWithError(w, http.StatusInternalServerError,
      err.Error())
      return
  }
```

```
  respondWithJSON(w, http.StatusCreated, user)
}
```

在创建了用户后，下面编写编辑用户的方法，并将其命名为 updateUser 方法。在该方法中，我们获取须进行修改的用户 ID，并转换以 JSON 形式接收的请求主体，然后调用 update 方法来修改数据实例。如果一切顺利，将发送对应的 HTTP 代码，如下所示：

```
func (a *App) updateUser(w http.ResponseWriter, r *http.Request) {
  vars := mux.Vars(r)
  id, err := strconv.Atoi(vars["id"])
  if err != nil {
    respondWithError(w, http.StatusBadRequest, "Invalid product ID")
    return
  }

  var user User
  decoder := json.NewDecoder(r.Body)
  if err := decoder.Decode(&user); err != nil {
    respondWithError(w, http.StatusBadRequest, "Invalid resquest
payload")
    return
  }
  defer r.Body.Close()
  user.ID = id

  if err := user.update(a.DB); err != nil {
    respondWithError(w, http.StatusInternalServerError, err.Error())
    return
  }

  respondWithJSON(w, http.StatusOK, user)
}
```

下面编写 CRUD 中的最后一个方法，即从数据库中移除用户的 deleteUser 方法。在该方法中，我们作为参数获取 id，并使用该数据创建用户实例。待该实例创建完毕后，将动用其 delete 方法。如果一切正常，将发送对应的 HTTP 代码，如下所示：

```
func (a *App) deleteUser(w http.ResponseWriter, r *http.Request) {
  vars := mux.Vars(r)
  id, err := strconv.Atoi(vars["id"])
  if err != nil {
    respondWithError(w, http.StatusBadRequest, "Invalid User ID")
    return
```

```
  }

  user := User{ID: id}
  if err := user.delete(a.DB); err != nil {
    respondWithError(w, http.StatusInternalServerError, err.Error())
    return
  }

  respondWithJSON(w, http.StatusOK, map[string]string{"result":
"success"})
}
```

最终，app.go 文件中的内容如下所示：

```
package main

import (
  "database/sql"
  "encoding/json"
  "fmt"
  "log"
  "net/http"
  "strconv"

  "github.com/codegangsta/negroni"
  "github.com/gorilla/mux"
  "github.com/jmoiron/sqlx"
    "github.com/lib/pq"
)
```

App 定义为包含应用程序配置值的结构，对应代码如下所示：

```
type App struct {
  DB                  *sqlx.DB
  Router          *mux.Router
}
```

Initialize 方法创建 DB 连接，并筹备所有的路由。对应代码如下所示：

```
func (a *App) Initialize(db *sqlx.DB) {

  a.DB = db
  a.Router = mux.NewRouter()
  a.initializeRoutes()
}
```

```
func (a *App) initializeRoutes() {
  a.Router.HandleFunc("/users", a.getUsers).Methods("GET")
  a.Router.HandleFunc("/user", a.createUser).Methods("POST")
  a.Router.HandleFunc("/user/{id:[0-9]+}",
   a.getUser).Methods("GET")
  a.Router.HandleFunc("/user/{id:[0-9]+}",
   a.updateUser).Methods("PUT")
  a.Router.HandleFunc("/user/{id:[0-9]+}",
   a.deleteUser).Methods("DELETE")
}
```

Run 方法负责初始化服务器，对应代码如下所示：

```
func (a *App) Run(addr string) {
  n := negroni.Classic()
  n.UseHandler(a.Router)
  log.Fatal(http.ListenAndServe(addr, n))
}

func (a *App) getUser(w http.ResponseWriter, r *http.Request) {
  vars := mux.Vars(r)
  id, err := strconv.Atoi(vars["id"])
  if err != nil {
    respondWithError(w, http.StatusBadRequest, "Invalid product ID")
    return
  }

  user := User{ID: id}
  if err := user.get(a.DB); err != nil {
    switch err {
      case sql.ErrNoRows:
          respondWithError(w, http.StatusNotFound, "User not found")
      default:
          respondWithError(w, http.StatusInternalServerError,
err.Error())
    }
    return
  }

  respondWithJSON(w, http.StatusOK, user)
}
```

```
func (a *App) getUsers(w http.ResponseWriter, r *http.Request) {
  count,   := strconv.Atoi(r.FormValue("count"))
  start,   := strconv.Atoi(r.FormValue("start"))

  if count > 10 || count < 1 {
    count = 10
  }
 if start < 0 {
               start = 0
 }

 users, err := list(a.DB, start, count)
 if err != nil {
   respondWithError(w, http.StatusInternalServerError, err.Error())
   return
 }

 respondWithJSON(w, http.StatusOK, users)
}

func (a *App) createUser(w http.ResponseWriter, r *http.Request) {
  var user User
  decoder := json.NewDecoder(r.Body)
  if err := decoder.Decode(&user); err != nil {
    respondWithError(w, http.StatusBadRequest, "Invalid request
payload")
    return
 }
 defer r.Body.Close()

 if err := user.create(a.DB); err != nil {
   fmt.Println(err.Error())
   respondWithError(w, http.StatusInternalServerError, err.Error())
   return
 }

 respondWithJSON(w, http.StatusCreated, user)
}

func (a *App) updateUser(w http.ResponseWriter, r *http.Request) {
  vars := mux.Vars(r)
  id, err := strconv.Atoi(vars["id"])
```

```
  if err != nil {
    respondWithError(w, http.StatusBadRequest, "Invalid product ID")
    return
  }

  var user User
  decoder := json.NewDecoder(r.Body)
  if err := decoder.Decode(&user); err != nil {
    respondWithError(w, http.StatusBadRequest, "Invalid resquest
payload")
    return
  }
  defer r.Body.Close()
  user.ID = id

  if err := user.update(a.DB); err != nil {
    respondWithError(w, http.StatusInternalServerError, err.Error())
    return
  }

  respondWithJSON(w, http.StatusOK, user)
}

func (a *App) deleteUser(w http.ResponseWriter, r *http.Request) {
  vars := mux.Vars(r)
  id, err := strconv.Atoi(vars["id"])
  if err != nil {
    respondWithError(w, http.StatusBadRequest, "Invalid User ID")
    return
  }

  user := User{ID: id}
  if err := user.delete(a.DB); err != nil {
    respondWithError(w, http.StatusInternalServerError, err.Error())
    return
  }

  respondWithJSON(w, http.StatusOK, map[string]string{"result":
"success"})
}

func respondWithError(w http.ResponseWriter, code int, message string)
```

```
{

  respondWithJSON(w, code, map[string]string{"error": message})
}

func respondWithJSON(w http.ResponseWriter, code int,
 payload interface{}) {
  response,   := json.Marshal(payload)

  w.Header().Set("Content-Type", "application/json")
  w.WriteHeader(code)
  w.Write(response)
}
```

代码看似篇幅较长，实际上，在微服务这一概念的基础上，对应内容十分简洁。

3．main.go 文件

目前，读者已对当前项目有了基本的了解，下面考察 main.go 文件。该文件负责将微服务操作所需的设置发送到应用程序中，并运行微服务本身。下列代码在开始阶段即实例化了数据库连接，该实例表示为每个应用程序所用的数据库。对应代码如下所示：

```
package main
import (
  "fmt"
  "github.com/jmoiron/sqlx"
    "github.com/lib/pq"
  "log"
  "os"
)

func main() {
  connectionString := fmt.Sprintf(
    "user=%s password=%s dbname=%s sslmode=disable",
    os.Getenv("APP DB USERNAME"),
    os.Getenv("APP DB PASSWORD"),
    os.Getenv("APP DB NAME"),
  )

  db, err := sqlx.Open("postgres", connectionString)
  if err != nil {
    log.Fatal(err)
  }
```

```
  a := App{}
  a.Initialize(cache, db)
  a.Run(":8080")
}
```

与 app.go 文件不同，main.go 文件较为简练。在 Go 语言中，全部应用程序均通过执行 main 函数进行初始化。在微服务中，情况也不例外。main 方法向应用程序实例发送连接数据库所需的内容。在 Run 方法结尾处，将在端口 8080 上运行应用程序服务器。

3.2 缓存策略

针对 Web 应用程序，需要制定一些缓存策略，这一类策略同样适用于微服务中。其中，最为常见的缓存策略是在查询操作后将信息存储在缓存中。图 3.2 表明，我们通过 API 接收请求，并在应用程序中对数据进行查询。第一次搜索在缓存中进行，如果缓存中未发现该数据，则搜索行为将在数据库中进行。当从数据库中返回查询结果时，对应值记录于缓存中。

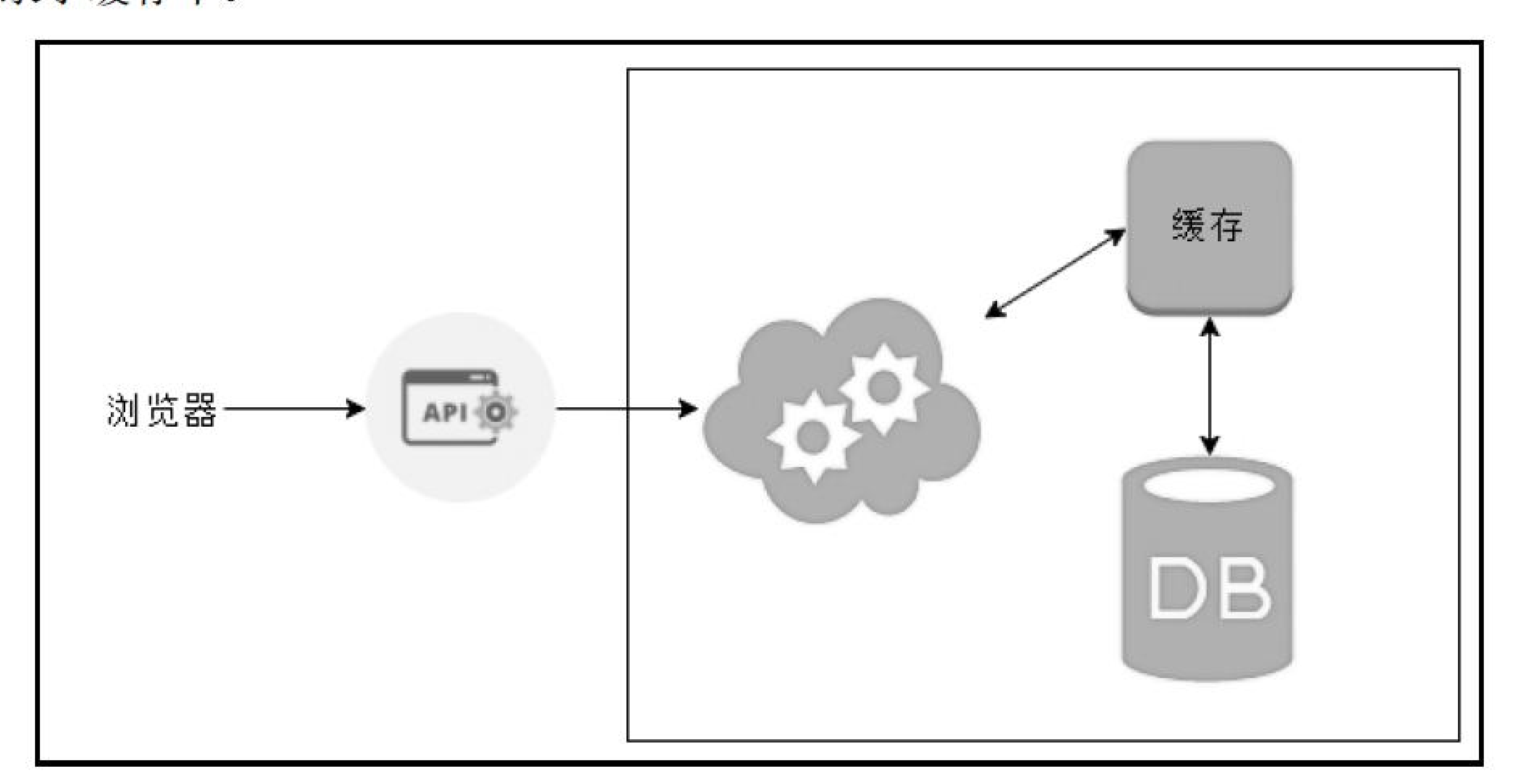


图 3.2

该策略可视为最简单的缓存机制。后续内容将在应用程序中对此加以调整并添加缓存层。

3.2.1 缓存机制的应用

第一步是使用 Redis 作为连接驱动程序下载依赖项，如下所示：

```
$ go get github.com/garyburd/redigo/redis
```

Redigo 则是基于 Redis 的通信接口，此处将采用 Redis 作为微服务的缓存工具。

接下来创建 cache/go 文件，该文件负责发送配置后的缓存实例。与其他曾创建的文件类似，此处需要声明一个工作包和依赖关系，如下所示：

```
package main

import (
  "log"
  "time"

  redigo "github.com/garyburd/redigo/redis"
)
```

随后，需要定义一个接口，进而创建 Redis 连接池，以及包含全部连接设置的一个结构。需要注意的是，连接池实例同样位于一个结构中。

下列代码定义了 Redis 连接池接口。

```
type Pool interface {
  Get() redigo.Conn
}
```

Cache 表示为包含缓存配置的结构，如下所示：

```
type Cache struct {

  Enable            bool

  MaxIdle           int

  MaxActive         int

  IdleTimeoutSecs   int

  Address           string

  Auth              string
```

```
  DB                  string
  Pool                *redigo.Pool
}
```

现在，我们将创建 struct 中的方法，对应方法负责提供一个新的连接池。

Redigo 包含了一个称之为 Pool 的结构，并在正确配置后返回所需内容。在当前缓存配置中，可启用 Enable 选项，并根据设置内容返回连接池。若未启用这一选项，那么将简单地忽略这一处理过程并返回 null。这意味着，将在最后阶段对连接池进行验证，若产生问题，则发出错误消息并停止服务器，然后提供相关服务，如下所示：

```
// NewCachePool return a new instance of the redis pool
  func (cache *Cache) NewCachePool() *redigo.Pool {
    if cache.Enable {
      pool := &redigo.Pool{
        MaxIdle: cache.MaxIdle,
        MaxActive: cache.MaxActive,
        IdleTimeout: time.Second * time.Duration(cache.IdleTimeoutSecs),
        Dial: func() (redigo.Conn, error) {
          c, err := redigo.Dial("tcp", cache.Address)
          if err != nil {
            return nil, err
          }
          if _, err = c.Do("AUTH", cache.Auth); err != nil {
            c.Close()
            return nil, err
          }
          if _, err = c.Do("SELECT", cache.DB); err != nil {
            c.Close()
            return nil, err
          }
          return c, err
        },
        TestOnBorrow: func(c redigo.Conn, t time.Time) error {
          _, err := c.Do("PING")
          return err
        },
      }
      c := pool.Get() // Test connection during init
      if _, err := c.Do("PING"); err != nil {
```

```
        log.Fatal("Cannot connect to Redis: ", err)
      }
      return pool
    }

  return nil
}
```

下面将定义相关方法以搜索缓存，并访问缓存中的数据。getValue 方法作为参数接收缓存中的搜索关键字；setValue 函数作为参数接收插入至缓存中的 key 以及 value，如下所示：

```
func (cache *Cache) getValue(key interface{}) (string, error) {
      if cache.Enable {
                  conn := cache.Pool.Get()
                  defer conn.Close()
                  value, err := redigo.String(conn.Do("GET", key))
                  return value, err
      }
      return "", nil
}

func (cache *Cache) setValue(key interface{}, value interface{}) error {
      if cache.Enable {
                  conn := cache.Pool.Get()
                  defer conn.Close()
                   , err := redigo.String(conn.Do("SET", key, value))
                  return err
      }
      return nil
}
```

通过这种方式，文件 cache.go 将被装载，以供当前应用程序使用。但是，在使用之前，需要对缓存文件进行一些适当调整。下面首先修改 main.go 文件。

在 main.go 文件中，可添加一些新的导入内容。当导入 flags 时，可直接从命令行中接收信息，并在 Redis 配置中对其加以使用，如下所示：

```
import (
  "flag"
  "fmt"
  "github.com/jmoiron/sqlx"
  _ "github.com/lib/pq"
```

```
  "log"
  "os"
)
```

当前需要执行的工作是添加传递至命令行中的选项，这一类修改操作也会出现于 main.go 文件中。首先，须创建 Cache 实例，随后添加该实例的指针，即设置项。如果命令行中未传递任何参数，那么，全部设置项均包含默认值。

相应地，设置的顺序如下所示。

（1）Adress：Redis 运行的位置。

（2）Auth：表示用于连接 Redis 的密码。

（3）DB：用作缓存的 Redis Bank。

（4）MaxIdle：处于空闲状态的最大连接数量。

（5）MaxActive：处于活动状态的最大连接数量。

（6）IdleTimeoutSecs：表示连接超时导致进入活动的时间。

在全部设置结束时，将利用 NewCachePool 方法生成一个新的连接池，并向缓存实例中传递指针，如下所示：

```
func main() {
  cache := Cache{Enable: true}
  flag.StringVar(
            &cache.Address,
            "redis address",
            os.Getenv("APP RD ADDRESS"),
            "Redis Address",
  )

  flag.StringVar(
            &cache.Auth,
            "redis auth",
            os.Getenv("APP RD AUTH"),
            "Redis Auth",
  )
  flag.StringVar(
            &cache.DB,
            "redis db name",
            os.Getenv("APP RD DBNAME"),
            "Redis DB name",
  )
  flag.IntVar(
            &cache.MaxIdle,
```

```
                "redis max idle",
                10,
                "Redis Max Idle",
  )

  flag.IntVar(
                &cache.MaxActive,
                "redis max active",
                100,
                "Redis Max Active",
  )
  flag.IntVar(
                &cache.IdleTimeoutSecs,
                "redis timeout",
                60,
                "Redis timeout in seconds",
  )
  flag.Parse()
  cache.Pool = cache.NewCachePool()
  ...
```

main.go 文件中的另一处修改则是向 Initialize 方法传递 App 缓存，如下所示：

```
...
a.Initialize(
  cache,
  db,
)
...
```

我们可适当地编辑 app.go 文件，并根据前述示意图（见图 3.2）高效地利用缓存。第一处修改位于 App 结构中，因为该结构中恰好存储了缓存，如下所示：

```
type App struct {
  DB         *sqlx.DB
  Router *mux.Router
  Cache Cache
}
```

目前，可确保 Initialize 方法接收缓存，并将对应值传递至 App 实例中，如下所示：

```
func (a *App) Initialize(cache Cache, db *sqlx.DB) {

  a.Cache = cache
  a.DB = db
```

```
  a.Router = mux.NewRouter()
  a.initializeRoutes()
}
```

至此，可在 App 的任意部分中使用缓存。下面修改 getUser 方法，并使用之前讨论的缓存结构。这里，需要对该方法中的两处内容进行修改，进而发挥内存机制的作用。

首先，无须在 PostgreSQL 中直接搜索数据，相应地，可检查数据是否已位于缓存中。如果数据已处于缓存中，则无须从数据库中获取数据。

其次，如果数据未处于缓存中，可在数据库中执行搜索，并在返回请求响应之前，将同一数据注册至缓存中。通过这种方式，在后续搜索中，数据将位于缓存中，且无须执行数据库上的查询操作，如下所示：

```
func (a *App) getUser(w http.ResponseWriter, r *http.Request) {
  vars := mux.Vars(r)
  id, err := strconv.Atoi(vars["id"])
  if err != nil {
    respondWithError(w, http.StatusBadRequest, "Invalid product ID")
    return
  }

  if value, err := a.Cache.getValue(id);err == nil && len(value)!= 0 {
    w.Header().Set("Content-Type", "application/json")
    w.WriteHeader(http.StatusOK)
    w.Write([]byte(value))
    return
  }

  user := User{ID: id}
  if err := user.get(a.DB); err != nil {
    switch err {
     case sql.ErrNoRows:
       respondWithError(w, http.StatusNotFound, "User not found")
     default:
       respondWithError(w, http.StatusInternalServerError, err.Error())
    }
    return
  }

  response,   := json.Marshal(user)
  if err := a.Cache.setValue(user.ID, response); err != nil {
    respondWithError(w, http.StatusInternalServerError, err.Error())
```

```
    return
  }

  w.Header().Set("Content-Type", "application/json")
  w.WriteHeader(http.StatusOK)
  w.Write(response)
}
```

通过上述修改工作，缓存将被查询值所替代，并可在不需要访问数据库的情况下进行响应。对于大多数场合来说，该方案已然足够。但在某些场合下，其中，请求负载较高，或者数据库要求较高，因此，即使采用了缓存机制，数据库仍有可能运行缓慢。针对这一类问题，接下来讨论另一种缓存策略。

3.2.2　缓存优先

缓存优先是一种非常有效的缓存策略，该策略将缓存置于数据库中的首要位置。需要说明的是，这一处理过程并不复杂。数据库中的全部操作主要在缓存中进行，并被发送至队列中；与此同时，相关操作开始使用该队列，并对数据库中的数据执行规范化操作，如图 3.3 所示。

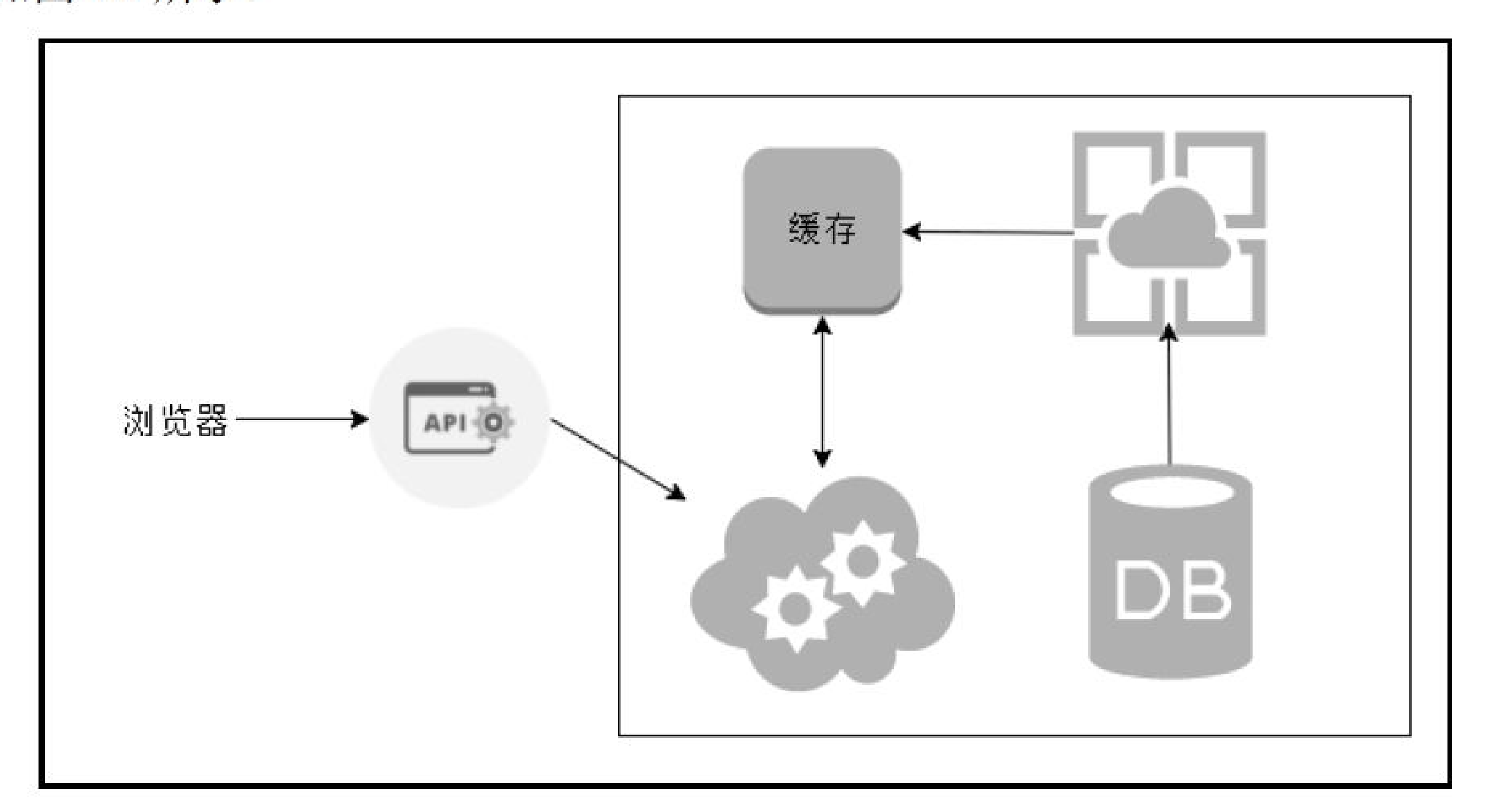


图 3.3

当实现这一技术方案时，在使用数据时须借助于队列和异步机制，并强烈推荐使用基于持久化的缓存机制。

下面将讨论缓存优先策略的实现方案。

3.2.3　队列任务

如前所述，我们希望将缓存用作一种“准数据库”。其中，所有信息均可被即刻加以使用，并于随后将数据高效地整合至数据库中。

下面通过以下几点内容，简要地了解一下持久化的内部流程。

（1）执行应用程序的请求。

（2）发送至应用程序的信息在 POST/PUT/DELETE 时在两个位置处进行注册。第一个位置是缓存，其中包含了请求中发送的全部信息；第二个位置是队列，其中包含了缓存信息的检索表。

（3）worker 检查队列中的内容。

（4）如果队列不为空，worker 将查询缓存中的数据。

（5）在搜索到缓存中的数据后，数据将在数据库中被持久化。

（6）此步骤仅适用于请求为 GET 的情况。在这种情况下，步骤（2）仅以缓存方式获取数据。如果在缓存中找不到数据，那么当前步骤执行数据库搜索。为了返回数据，搜索被记录在缓存中。为了返回最终数据，当前搜索结果被记录于缓存中。

需要着重指出的是，仅仅发送到缓存和队列就足以将消息返回给客户端，对应结果表示为一条成功或失败信息，如图 3.4 所示。

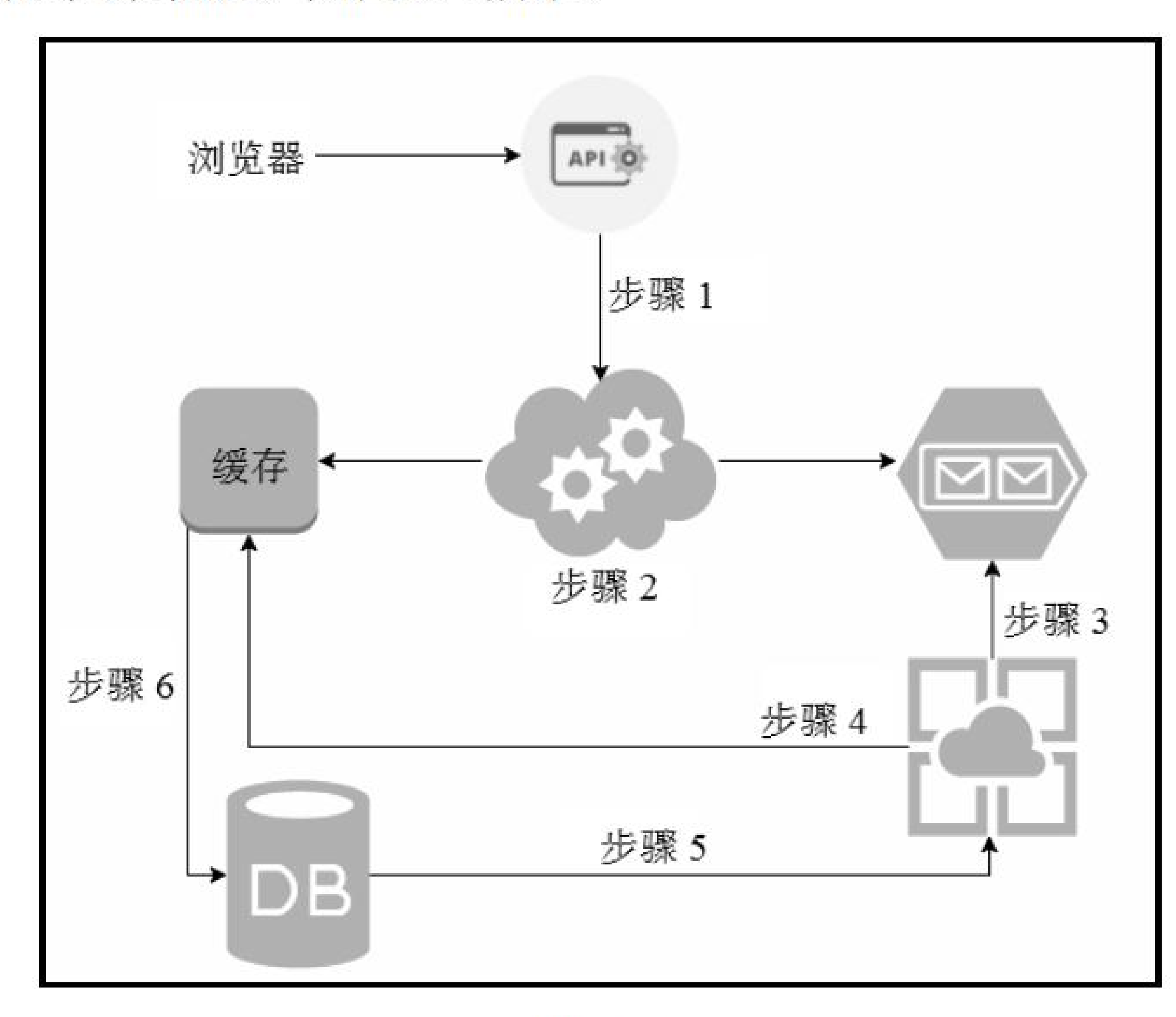


图 3.4

读者在了解了上述概念后，下面即可着手修改代码并对其加以运用。main.go 文件中定义了当前应用程序以及设置项。在该文件中，将声明处理过程中的队列名。下列代码定义了相关的常量。

```
const (
  createUsersQueue = "CREATE USER"
  updateUsersQueue = "UPDATE USER"
  deleteUsersQueue = "DELETE USER"
)
```

上述常量用于应用程序中的其他部分，以向队列发送数据。

```
func (cache *Cache) enqueueValue(queue string, uuid int) error {
  if cache.Enable {
             conn := cache.Pool.Get()
             defer conn.Close()
              , err := conn.Do("RPUSH", queue, uuid)
             return err
  }
  return nil
}
```

下一个步骤是调整 app.go 文件中的 createUser 方法，该方法将移除源自数据库中的持久化方面的内容。此处唯一的访问操作来自 PostgreSQL，用于获取应用于实体实例上的新序列。

在指定了 ID 实体后，通过 ID 作为键，并以 JSON 格式的实体作为值，即可在缓存中进行注册。随后，可将该键发送至 CREATE_USER 队列中。考察下列代码示例：

```
func (a *App) createUser(w http.ResponseWriter, r *http.Request) {
     var user User
     decoder := json.NewDecoder(r.Body)
     if err := decoder.Decode(&user); err != nil {
                respondWithError(w, http.StatusBadRequest, "Invalid
request payload")
                return
     }
     defer r.Body.Close()

     // get sequence from Postgres
     a.DB.Get(&user.ID, "SELECT nextval('users id seq')")

     JSONByte, _ := json.Marshal(user)
```

```
     if err := a.Cache.setValue(user.ID, string(JSONByte)); err != nil {
                 respondWithError(w, http.StatusInternalServerError,
err.Error())
                 return
     }

     if err:=a.Cache.enqueueValue(createUsersQueue,user.ID);err!=nil
{
                 respondWithError(w, http.StatusInternalServerError,
err.Error())
                 return
     }

     respondWithJSON(w, http.StatusCreated, user)
}
```

对于队列中的数据，以及数据库记录基础上的缓存机制，如果此时执行查询操作，数据将从缓存中返回，而不是数据库。

下一个步骤是在数据库缓存中注册信息，稍后将对此加以讨论。

3.2.4 异步机制和 worker

目前，相关信息已位于队列中，但尚未存储于数据库中，其原因在于，我们尚未使用队列中的信息，仅是将其发送至数据库而已。

回复队列中的数据，以及将数据发送至规范化数据库中的处理行为须处于异步状态。最终，同一软件中似乎存在两个应用程序。其中，第一个程序负责接收数据，而第二个程序则负责处理数据。

下面考察上述处理过程在应用程序代码中的实现方式。首先，需要创建一个名为 workers.go 的文件；随后，还需要声明一个工作包，如下所示：

```
package main

import (
  "encoding/json"
  redigo "github.com/garyburd/redigo/redis"
  "github.com/jmoiron/sqlx"
  "log"
  "sync"
)
```

下面将定义一个结构，进而满足所有 worker 的相关操作。该结构包含了实例缓存设置、worker 的数据库实例 ID，以及 worker 使用的队列。

下列代码定义了 Worker 结构，其中包含了 worker 的配置值。

```
type Worker struct {
  cache Cache
  db        *sqlx.DB
  id        int
  queue string
}
```

newWorker 函数负责初始化 worker，并作为 worker 结构的参数接收相关数据，如下所示：

```
func newWorker(id int, db *sqlx.DB, cache Cache,
queue string) Worker {
  return Worker{cache: cache, db: db, id: id, queue: queue}
}
```

下一步是定义 worker 中的 process 方法，该方法将针对队列进行操作，并向数据库发送数据。如果该过程失败，process 方法将重新发送队列中的数据。

process 方法接收 worker 的 ID。在该方法中，设置了一个无限循环（向连接池请求缓存连接），同时还声明了两个变量，即 channel 和 uuid。这里，两个变量利用队列中的信息进行填充。在当前示例中，通道的存在仅为实现订阅 Redigo API，对应变量为 uuid。

Redis 中的 BLPOP 函数负责对队列进行操作。当使用 uuid 时，可通过 Redis 中的 GET 函数获取缓存中的数据。在所检索信息的基础上，可通过对应结构实例化一个用户。随后，将调用用户实例的 create 方法在数据库中注册新的用户，如下所示：

```
func (w Worker) process(id int) {
  for {
    conn := w.cache.Pool.Get()
    var channel string
    var uuid int
    if reply, err := redigo.Values(conn.Do("BLPOP", w.queue,
     30+id)); err == nil {

      if  , err := redigo.Scan(reply, &channel, &uuid); err != nil {
        w.cache.enqueueValue(w.queue, uuid)
        continue
      }

      values, err := redigo.String(conn.Do("GET", uuid))
```

```
      if err != nil {
        w.cache.enqueueValue(w.queue, uuid)
        continue
      }

      user := User{}
      if err := json.Unmarshal([]byte(values), &user); err != nil {
        w.cache.enqueueValue(w.queue, uuid)
        continue
      }

      log.Println(user)
      if err := user.create(w.db); err != nil {
        w.cache.enqueueValue(w.queue, uuid)
        continue
      }

    } else if err != redigo.ErrNil {
        log.Fatal(err)
    }
    conn.Close()
  }
}
```

下面在 workers.go 文件中定义 UsersToDB 函数。该函数创建队列的 worker 数量，并采用异步方式实例化、初始化 worker。

UsersToDB 创建 worker 并使用队列，如下所示：

```
func UsersToDB(numWorkers int, db *sqlx.DB, cache Cache,
  queue string) {
    var wg sync.WaitGroup
    for i := 0; i < numWorkers; i++ {
          wg.Add(1)
          go func(id int, db *sqlx.DB, cache Cache, queue string) {
               worker := newWorker(i, db, cache, queue)
               worker.process(i)
               defer wg.Done()
          }(i, db, cache, queue)
  }
  wg.Wait()
}
```

workers.go 文件的最终结果如下所示：

```
package main

import (
  "encoding/json"
  redigo "github.com/garyburd/redigo/redis"
  "github.com/jmoiron/sqlx"
  "log"
  "sync"
)
```

Worker 表示包含 worker 配置值的结构，如下所示：

```
type Worker struct {
  cache Cache
  db      *sqlx.DB
  id      int
  queue string
}
```

UsersToDB 负责创建 worker 并使用队列，如下所示：

```
func UsersToDB(numWorkers int, db *sqlx.DB, cache Cache,
  queue string) {
    var wg sync.WaitGroup
    for i := 0; i < numWorkers; i++ {
      wg.Add(1)
      go func(id int, db *sqlx.DB, cache Cache, queue string) {
        worker := newWorker(i, db, cache, queue)
        worker.process(i)
        defer wg.Done()
      }(i, db, cache, queue)
    }
    wg.Wait()
}

func newWorker(id int, db *sqlx.DB, cache Cache,
    queue string) Worker {
    return Worker{cache: cache, db: db, id: id, queue: queue}
}

func (w Worker) process(id int) {
    for {
      conn := w.cache.Pool.Get()
      var channel string
```

```
    var uuid int
    if reply, err := redigo.Values(conn.Do("BLPOP", w.queue,
      30+id)); err == nil {

        if _, err := redigo.Scan(reply, &channel, &uuid); err != nil {
          w.cache.enqueueValue(w.queue, uuid)
          continue
        }

        values, err := redigo.String(conn.Do("GET", uuid))
        if err != nil {
          w.cache.enqueueValue(w.queue, uuid)
          continue
        }

        user := User{}
        if err := json.Unmarshal([]byte(values), &user); err != nil {
          w.cache.enqueueValue(w.queue, uuid)
          continue
        }

        log.Println(user)
        if err := user.create(w.db); err != nil {
          w.cache.enqueueValue(w.queue, uuid)
          continue
        }

      } else if err != redigo.ErrNil {
          log.Fatal(err)
      }
      conn.Close()
    }
}
```

然而，在实际调用 worker 时，还需要在 main.go 文件中稍作编辑。回忆一下，main.go 文件负责初始化微服务。其中，主函数中新添加了两项内容。除此之外，还可通过命令行以及队列接收 worker 数量，并初始化应用程序，同时使用前缀 go 初始化 worker 自身，如下所示：

```
func main() {
  var numWorkers int
  cache := Cache{Enable: true}
  flag.StringVar(
```

```
    &cache.Address,
    "redis address",
    os.Getenv("APP RD ADDRESS"),
    "Redis Address",
)
flag.StringVar(
    &cache.Auth,
    "redis auth",
    os.Getenv("APP RD AUTH"),
    "Redis Auth",
)
flag.StringVar(
    &cache.DB,
    "redis db name",
    os.Getenv("APP RD DBNAME"),
    "Redis DB name",
)
flag.IntVar(
    &cache.MaxIdle,
    "redis max idle",
    10,
    "Redis Max Idle",
)
flag.IntVar(
    &cache.MaxActive,
    "redis max active",
    100,
    "Redis Max Active"
)
flag.IntVar(
    &cache.IdleTimeoutSecs,
    "redis_timeout",
    60,
    "Redis timeout in seconds"
)
flag.IntVar(
    &numWorkers,
    "num workers",
    10,
    "Number of workers to consume queue"
)
flag.Parse()
cache.Pool = cache.NewCachePool()
```

```
  connectionString := fmt.Sprintf(
      "user=%s password=%s dbname=%s sslmode=disable",
      os.Getenv("APP DB USERNAME"),
      os.Getenv("APP DB PASSWORD"),
      os.Getenv("APP DB NAME"),
  )

  db, err := sqlx.Open("postgres", connectionString)
  if err != nil {
      log.Fatal(err)
  }

  go UsersToDB(numWorkers, db, cache, createUsersQueue)
  go UsersToDB(numWorkers, db, cache, updateUsersQueue)
  go UsersToDB(numWorkers, db, cache, deleteUsersQueue)

  a := App{}
  a.Initialize(cache, db)
  a.Run(":8080")
}
```

最终，当前应用程序将采用缓存优先策略。在 UsersService 中，该策略工作良好，其原因在于，用户 ID 易于获取和计算。当前，好消息是所需的一切内容都已在应用程序中设置完毕，且无须再向栈中添加新内容。尽管如此，某些缺陷依然存在。

假设，检索缓存的 ID 难于获取、计算，对此，还可考虑标识键组合。这里的问题是，如何解决此类障碍以实现缓存优先策略？

对于此类问题，较好的方法是命令查询职责分离（CQRS）。当搜索方案和维护工作变得越发复杂时，CQRS 则是一种较好的选择方案。

3.3 节将对该方案加以讨论。

3.3 CQRS——查询策略

CQRS 是我们需要了解的一个较为重要的概念。前述内容一直在强调，每种架构都包含了一个工具箱，CQRS 即是其中的一种工具。

3.3.1 CQRS 的概念

CQRS 是命令查询职责分离的简称。顾名思义，这一概念与数据读、写职责分离相

关。注意，CQRS 是一种代码模式，而非架构模式。

下面考察日常生活中的一类经典场景，并考虑如何将 CQRS 应用于其中。

随着互联网的不断发展，一般不太可能针对少数用户创建应用程序，大多数应用程序具备可扩展性、高性能以及可用性等特征。那么，应用程序应如何与大量的用户实现较好的同步协同工作呢？创建满足此类需求的模型往往十分复杂，此时，数据库往往会成为瓶颈。

作为一个例子，下面考察金融信用体系，消费者可以在此购物过程中获得快速信贷。其间，针对数据更改的访问操作有时较为轻松，而一些时候则会变得十分紧张。

此类问题的答案在于在 *n* 台服务器中扩展应用程序。针对于此，我们可迁移至云计算平台上，并根据需要创建脚本自动扩展机制。

应用程序的可扩展概念能够解决某些实用性问题，例如同时支持多个用户，而不会对应用程序的性能产生影响。

那么，扩展应用程序服务器是否能解决所有问题？

死锁、超时以及运行缓慢均意味着数据库可能负担了过多的需求任务。

增加应用程序实例的数量并不能保证其始终可用。在当前示例中，应用程序完全依赖于数据库的有效性。

与扩展应用服务器相比，数据库的扩展行为可能要复杂得多，其成本也要高得多。通常，正是由于数据库的过度消耗，应用程序才会出现性能问题。

可以执行复杂的查询来获取数据库数据。ORM 可以通过映射实体和表连接操作来过滤数据——数据过滤过程的复杂性也随之增加。

此外，内容过时（obsolescence）这种现象确实存在。有限的数据集经常被大量用户查阅和修改。这意味着，屏幕上显示的一个数据可能已经被另一个数据所修改。因此，可以规定所显示的全部信息可能均已过时。

3.3.2 理解 CQRS

当多个服务器使用单一数据库并进行读、写操作时，这往往会在数据操控过程中导致多个瓶颈的出现，并产生各种性能问题。另外，获取显示数据的业务规则处理过程将会占用额外的处理时间。最后，我们还是需要考虑显示数据的过时问题。

针对数据的读、写职责，CQRS 体现了一种划分机制，这两个概念采用了不同的物理存储，同时也意味着，存在着独立的数据记录和检索方式。相应地，查询操作在独立的规范化数据库中以同步方式完成；而写入行为则针对规范化数据库以异步方式实现。在概念层次上，缓存优先策略仍可视作一类 CQRS 实现类型。

实际上，除了优化之外，CQRS 无须在应用程序的各个处理阶段加以使用。基于 DDD 的边界上下文建模（参见 http://www.microsofttranslator.com/bv.aspx?from=ptto=enr=truea=http%3A%2F%2Fwww.eduardopires.net.br%2F2016%2F03%2Fddd-bounded-context%2F）可实现 CQRS，而其他方案则无能为力。

根据应用程序的实际需求，CQRS 的实现过程可能非常简单，或者是异常复杂。无论采取哪一种实现方式，CQRS 总会带来额外的复杂度，因而在使用该模式之前需要对当前方案予以评估。其中，命令负责调整数据库中数据的状态；而查询则负责从数据库中检索信息。

我们可以将此视为分离的 CommandStack 职责，以及 *n* 层架构中的 QueryStack。

- QueryStack 相对简单，负责检索即将显示的数据。可以说，QueryStac 是一类同步层，并从非正规读取机制中检索数据。

对应数据库可视为 NoSql 类型的 MongoDB（参见 https://www.microsofttranslator.com/bv.aspx?from=ptto=ena=https%3A%2F%2Fwww.mongodb.com%2F),Redis(https://www.microsofttranslator.com/bv.aspx?from=ptto=ena=http%3A%2F%2Fredis.io%2F）、RavenDB（参见 https://www.microsofttranslator.com/bv.aspx?from=ptto=ena=https%3A%2F%2Fravendb.net%2F），或者是市场上其他类型的数据库。这里，"非常规"这一概念是指，可以针对每幅视图应用一张表；或者作为简单查询返回全部需要显示的数据。

- CommandStack。CommandStack 有可能处于异步状态，其中包含了实体、业务规则以及其他处理过程。考虑 DDD 这一情形，该领域即属于应用程序的 CommandStack 中。CommandStack 遵循以行为为中心的方案，其中，全部业务意向最初由客户端所触发。相应地，我们使用命令这一概念表达业务意向。具体来说，所声明的命令采用了命令形式，并以事件方式异步出现；通过 CommandHandlers 予以解释并返回成功或失败消息。

当命令被触发，并在写入过程中修改实体状态时，应针对代理创建一个数据库处理进程，进而更新读取操作中所需的数据。

对于同步机制，以下内容列举了读取操作和记录的同步策略，读者可针对具体场合选择最佳方案。

- 自动更新。记录的数据库状态变化将产生一个同步处理进程，以实现更新操作。
- 可能的更新操作。记录的数据库的所有状态变化将触发一个异步处理进程，并更新读取数据库，进而实现最终的数据一致性特征。
- 受控更新操作。产生一个常规处理进程和进度表，以对数据库实现同步化操作。
- 按需更新。每个队列经与记录比较后，检测读取数据库时的一致性，并在内容过期后强制更新。

任何更新行为都是最常用的策略之一，因为它假定任何给定的显示数据都可能已经过期，所以没有必要强制执行同步更新过程。

许多 CQRS 实现可能需要消息代理来处理命令和事件。在这种情况下，图 3.5 展示了一种方案。

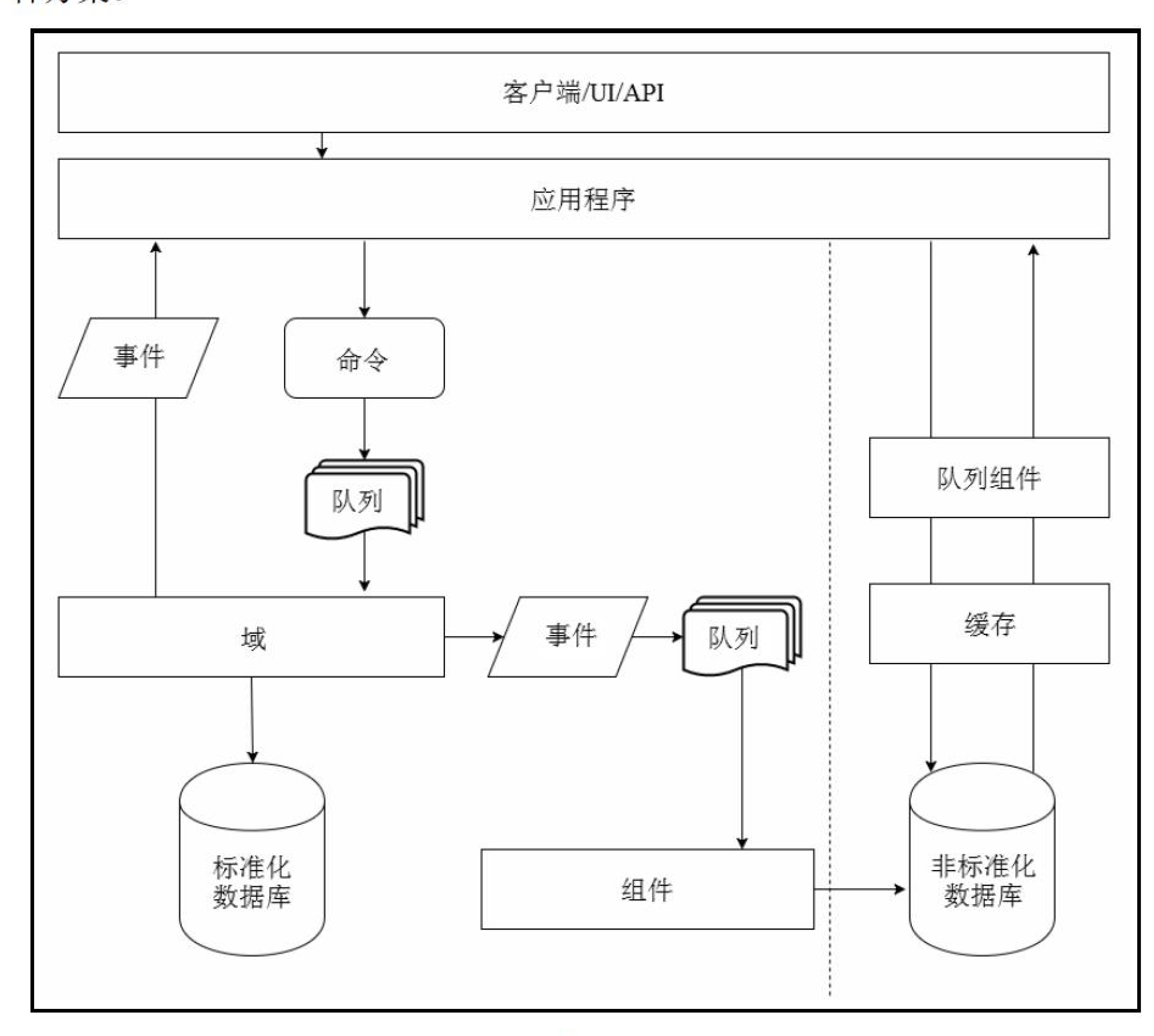


图 3.5

3.3.3　CQRS 的优点和缺陷

CQRS 提出了与经典的单体系统不同的概念。在单体系统中，写入和读取的整个过程历经相同的层，并且在处理业务规则和数据库使用方面相互竞争。

与 CQRS 相关的概念为我们提供了较好的可扩展性以及实用性方面的内容，其优势主要体现在以下方面：

- 相关命令处于异步状态，并在队列中加以处理，进而降低等待时间。

- ❑ 读、写操作针对同一资源不再处于竞争状态。
- ❑ QueryStack 上的查询操作彼此分离且相互无关，且不依赖于 CommandStack 处理机制。
- ❑ 可以分别对 CommandStack 处理进程和 QueryStack 处理进程进行扩展。
- ❑ 在业务意图和其他 DDD 概念中，使用常见的语言展现富有表现力的领域表达结果。

对于应用程序来说，CQRS 包含诸多优点，但也包含了某些缺陷，如下所示：

- ❑ CQRS 相对复杂。对于应用程序、针对某个领域的清晰理解，以及常见语言，其复杂度均有所提升。
- ❑ 在使用最终一致性模型时，应对此予以格外关注。虽然这一概念并非是强制性的，但应对其加以谨慎处理。
- ❑ 取决于具体实现，尤其是使用最终一致性策略时，一般会采用消息代理机制。然而，这增加了应用程序的复杂性以及组件监控的困难。

需要注意的是，CQRS 并非是一种架构模式，它可以理解为是一种应用程序的组件化形式。一种常见的错误观念是，CQRS 可与事件源连同使用。事件源与 CQRS 联系紧密，但却可以实现独立于 CQRS 的事件源。

不可否认，CQRS 是一种较好的模式，应在任意类型的应用程序中引起足够的重视，特别是微服务。扩展的灵活性以及高可用性远远超出了该模式所带来的额外的复杂度。

3.4　事件源——数据完整性

在进入事件源这一话题之前，有必要了解一下大多数标准应用程序的相关功能。

当在数据库中运行 UPDATE 命令时，我们都会对数据库中的当前表示进行修改。据此，无论何时在数据库中执行查询操作，都将搜索某个记录的当前状态。这一行为称作状态变化。下面通过一个示例对此加以解释。

假设我们有一个表，负责管理用户的访问级别，如下所示：

ID	user_name	status	user_manager
1	John Doe	admin	Manager1

在该表中，用户 John Doe 为管理员，其状态表示为 Manager1。一段时间以后，发现 John Doe 不可能一直是应用程序的管理员，并且对表中的这一行执行了 UPDATE 操作，如下所示：

```
UPDATE status user set status='normal user', user manager='Manage2' WHERE
ID=1;
```

这一变化将导致针对 ID = 1 的修改操作，如下所示：

ID	user_name	status	user_manager
1	John Doe	normal_user	Manager2

其中，Manager2 负责修改用户的状态。用户 John Doe 的当前状态表示为 normal_user。那么，之前的状态又是什么？在一段时间内，谁来负责让 John Doe 的身份成为管理员？

在数据库中，记录修改的默认行为使我们无法了解应用程序中记录的状态——一般仅知晓记录表的当前状态。对此，我们需要知道进入事件源的记录的历史信息。

事件源这一概念则稍有不同。数据库当前状态中的每次变化都是流中的一个新事件；表中的每次更新操作都会生成一行，即状态的更改。如果更改为错误的状态，那么，将对该状态进行修改并生成新行。因此，我们将持有全部变化内容，直至到达用户权限时间。

从开始阶段，status_user 表中即使用了事件源这一概念，对应记录如下所示：

ID	user_name	status	user_manager
1	John Doe	admin	Manager1
1	John Doe	normal_user	Manager2

当用户 John Doe 采用新状态时，将跟踪当前变化并生成新的一行。对于数据库记录来说，事件源仅使用 append only 模式。

事件源也包含了自身的优缺点。其优点主要体现在基于历史操作的记录维护功能；而记录的编辑操作往往呈指数级增长，这也是事件源的一个缺点。对此，一种较为常见的做法是利用 CQRS 查看事件源，而不是根据大量的同一记录阻塞数据库的搜索操作。

3.5　本章小结

本章开始涉及编程方面的内容，包括一些基本的概念的高级缓存机制。另外，本章还讨论了模式、CQRS 以及事件源的具体行为和相关功能。

后续内容将继续对当前应用程序加以构建。

第 4 章将围绕微服务编码展开讨论，并着手实现环境配置。

第 4 章　微服务生态环境

在软件开发过程中，总会出现一些我们所不了解的内容。例如，软件最终是否会成功运行？当编写应用程序并将其置入产品中时，也会产生各种问题，其间也会伴随着失败。

有人曾指出，零 bug 的软件是不存在的。充其量，软件只是存在未知的 bug。这一说法并非谬论，甚至是 100%正确的。

通常，应用程序包含较高的测试覆盖率；另外，领域业务还涉及自动化测试以及集成测试。显然，一切工作良好。但是，当谈及微服务时，还会涵盖一些潜在的风险，例如网络连接、负载平衡中的错误，以及外部服务使用过程中的故障。

微服务中的问题可能源自开发团队中产生的 bug，或者是与其他服务集成后造成的不良后果。毫无疑问，应用程序中的缺陷将导致产品的一种良性失败。

这里，读者可能会产生疑问，难道还会存在一种良性失败？答案是肯定的。

良性失败意味着，问题不包含不可用服务，或者无效内容仅呈现为局部特性，而不涉及整个系统。除此之外，快速发现问题同样是一种“成功”的标志。

对于故障的良性结果，我们将采取一些措施，并在工作堆栈中解决此类问题。本章主要涉及以下话题：

- 容器中的分离机制。
- 数据分布。
- 对故障的反应机制。

4.1　容器中的分离机制

为了阻止故障的发生，重要的是需要了解产生故障的方式。下面考察单体应用程序中较为常见的操作，此类也会出现于微服务架构中，进而有助于我们理解故障前的响应方式。

一种较为常见的方法是将应用程序的整体结构置于单一存储库中。也就是说，软件代码、缓存以及应用程序的所有其他特性均位于同一台机器设备中，这种情况屡见不鲜。

图 4.1 显示了缓存、数据库、API 以及业务逻辑层位于同一位置时的情形。初看之下，

这一配置行为并无问题。在同一台机器中，诸如延迟、包丢失和部署复杂性等问题都被简化了。

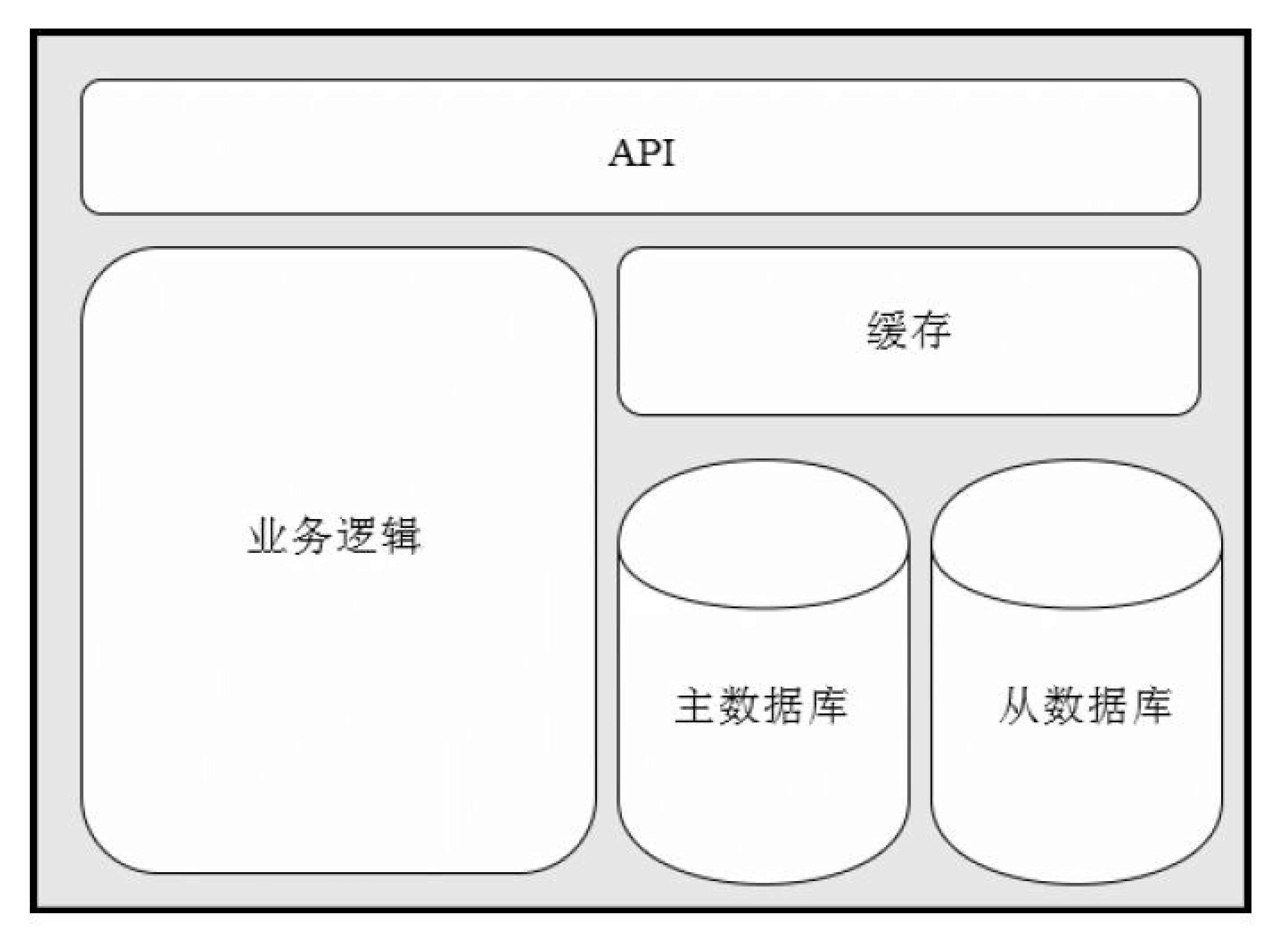


图 4.1

在当前方案中，假设容器出现故障，那么，将难以分辨容器故障所对应的组件。这一辨识过程将占用大量的时间。另外，缓存中的某个缺陷可能会导致应用程序崩溃。故障的产生过程往往是以渐进方式进行的，当意识到容器的系统故障时，可能为时已晚。

由于所有组件均处于连接状态，并且无法对其进行单独处理，因而整个应用程序将被重新启动。针对此类行为，尚不存在一种优雅的系统可对此提供支持。除了导致某种程度的不一致结果之外，还可能丢失数据。

之前的一些积极因素现在被证明是一种很容易破坏弹性的选择方案，例如部署复杂性的降低，以及延迟和包丢失的减少。将整个应用程序堆栈保存在一个物理组件中，意味着所有软件层都将使用组件的相同特性。此时，故障的缓解往往与自身方案的选取有关。

这种系统故障不仅影响了应用程序的可用性，而且还可能影响产品在客户和投资者眼中的可信度。图 4.2 显示了缓存出现错误且数据库过载时发生的故障，进而导致系统崩溃。

这依然不是最糟糕的情况。对于应用程序来说，最不希望听到的即是“成功”一词，成功意味着应用程序的扩展。也就是说，在不提升弹性的情况下，成倍地增加可用性。高可用性将与所支持的命中数量有关，而不是保持可用性，即使发生内部故障。

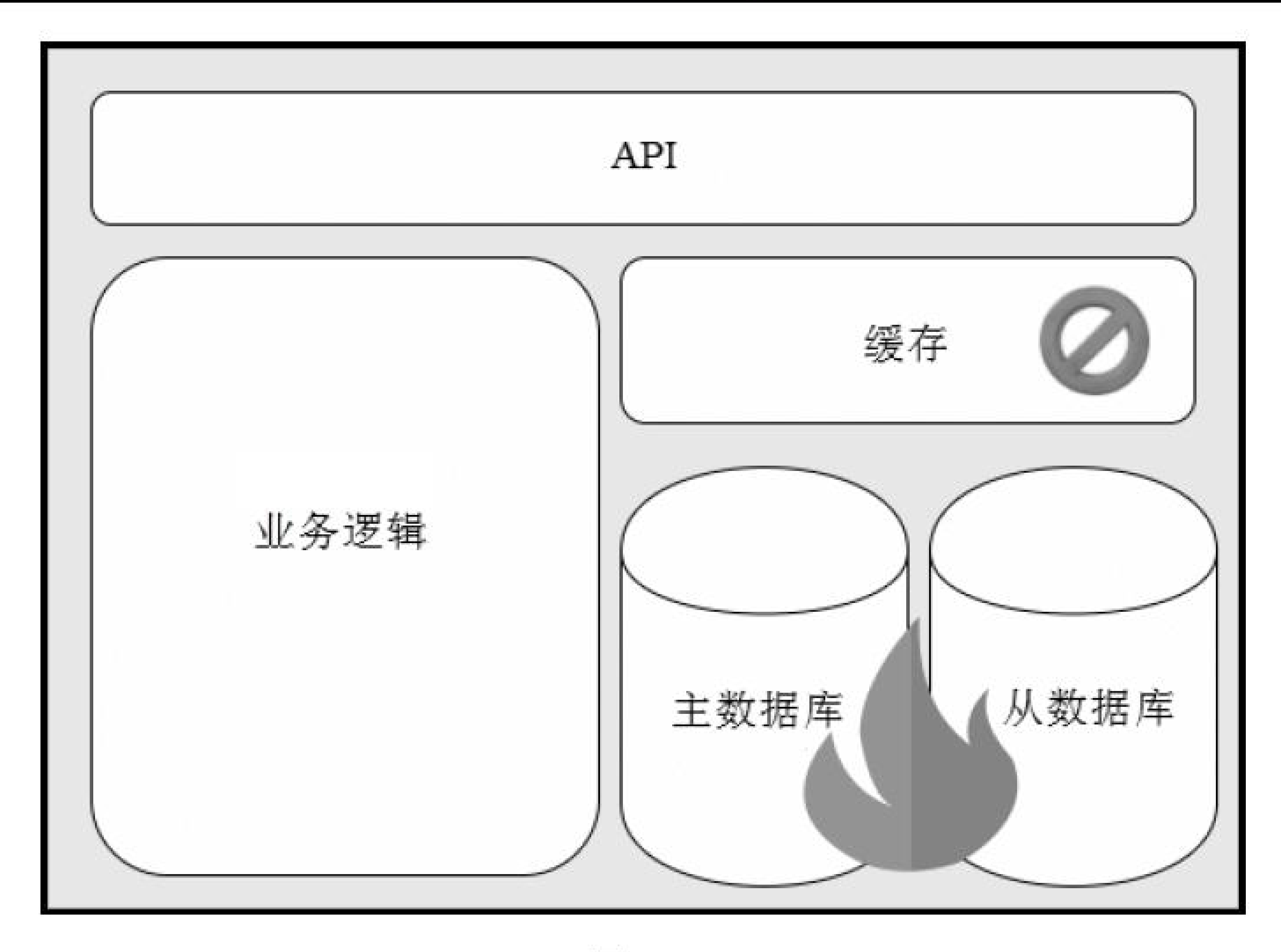


图 4.2

这可能并不是当前应用程序的选择方案。通常，在较差的微服务生态系统中，单体架构呈现为重复状态。对于一些软件工程师来说，微服务架构仅仅是将业务逻辑与应用程序分离开来。这种认知是错误的——应用程序的组件也必须以可伸缩的方式进行设计，因此，在极端情况下，必须可实现快速修复或重置行为。

4.1.1　分层服务架构

对于应用程序所采取的分离机制，这是一种十分常见的反模式。一些工程师常会在堆栈组件的分离与逻辑层的分离之间产生混淆，这样就产生了一个缺少业务表达的贫血域。

图 4.3 显示了一种较差的实现方式。其中，软件由 3 个不需要的物理组件组成。同时，应用程序的分离具有较细的粒度。此外，编排器和数据访问并不具备实际意义。

基本的业务内容需要对数据库进行访问。很快，构建这种分离机制便没有逻辑可言了。另一点需要注意的是，如果编排器直接访问数据，也就意味着，它在应用业务方面具有一定的智能。另外，编排器也是完全不必要的——仅存在一个业务层，因而并不存在数据可供编排。

另外，当前粒度是微服务架构中非常常见的错误，同时也是我们不惜一切代价所要避免的。该粒度会提升系统的复杂度，同时增加维护工作物理组件的成本，以及多层间的通信成本。

下一步是在应用程序中构建一种具有实际意义的分离机制。对此，我们的脑海中应浮现出一幅与图 4.3 类似的结构图，旨在避免一系列的重复行为。

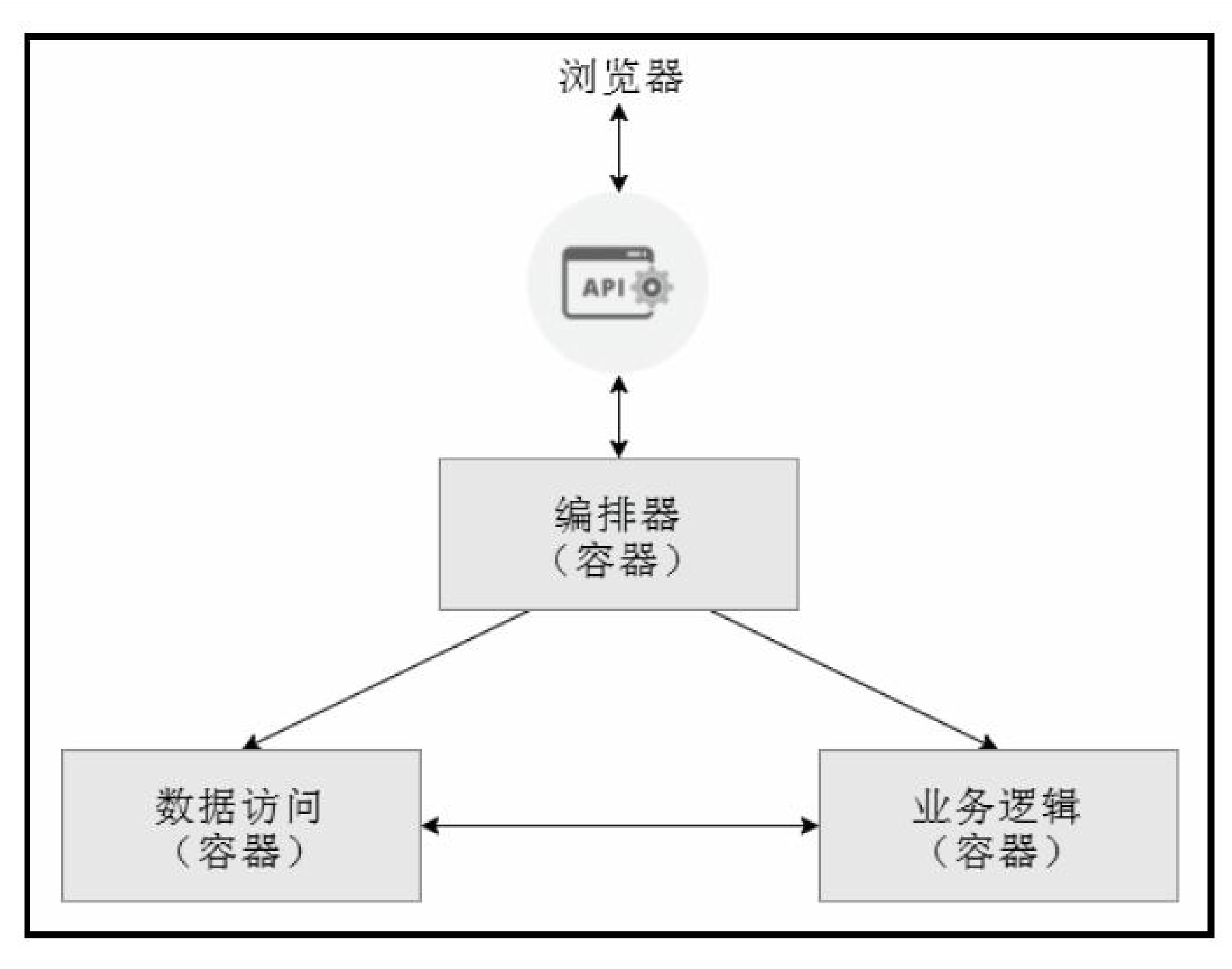


图 4.3

4.1.2　分离 UsersService

微服务组件的分离可通过多种方式实现。如果采用物理或虚拟容器，对于应用程序来说，分离机制应以一种较为健康的方式进行，从而支持可扩展性、弹性、可用性，以及版本化微服务中的各方面内容。

这里强烈推荐使用 Docker 来划分组件。首先，Docker 很容易自动化操作；其次，Docker 可方便地在其他环境中复制同一生产堆栈，包括开发权，从而避免令人不愉快的意外情况发生。

在我们的所有微服务中，将使用 Docker 作为应用程序容器的生成工具。本书将对 Docker 的应用予以全面的介绍，同时也包括它在微服务中的适用性。

在开始对 Docker 进行编码之前，下面首先展示一下组件分离后应用程序的外观，如图 4.4 所示。

图 4.4 显示了微服务的实际现状。因此，我们还会针对当前项目进行适当的调整。

1. 创建 Dockerfile

这里首先在 UsersService 创建 Dockerfile 文件，该文件负责组装容器的映像并编译应用程序。

FROM 策略表明应用程序使用哪一种操作系统。在当前情况下，将使用 golang:latest，这是一个安装了 Go 语言的 Ubuntu 系统。

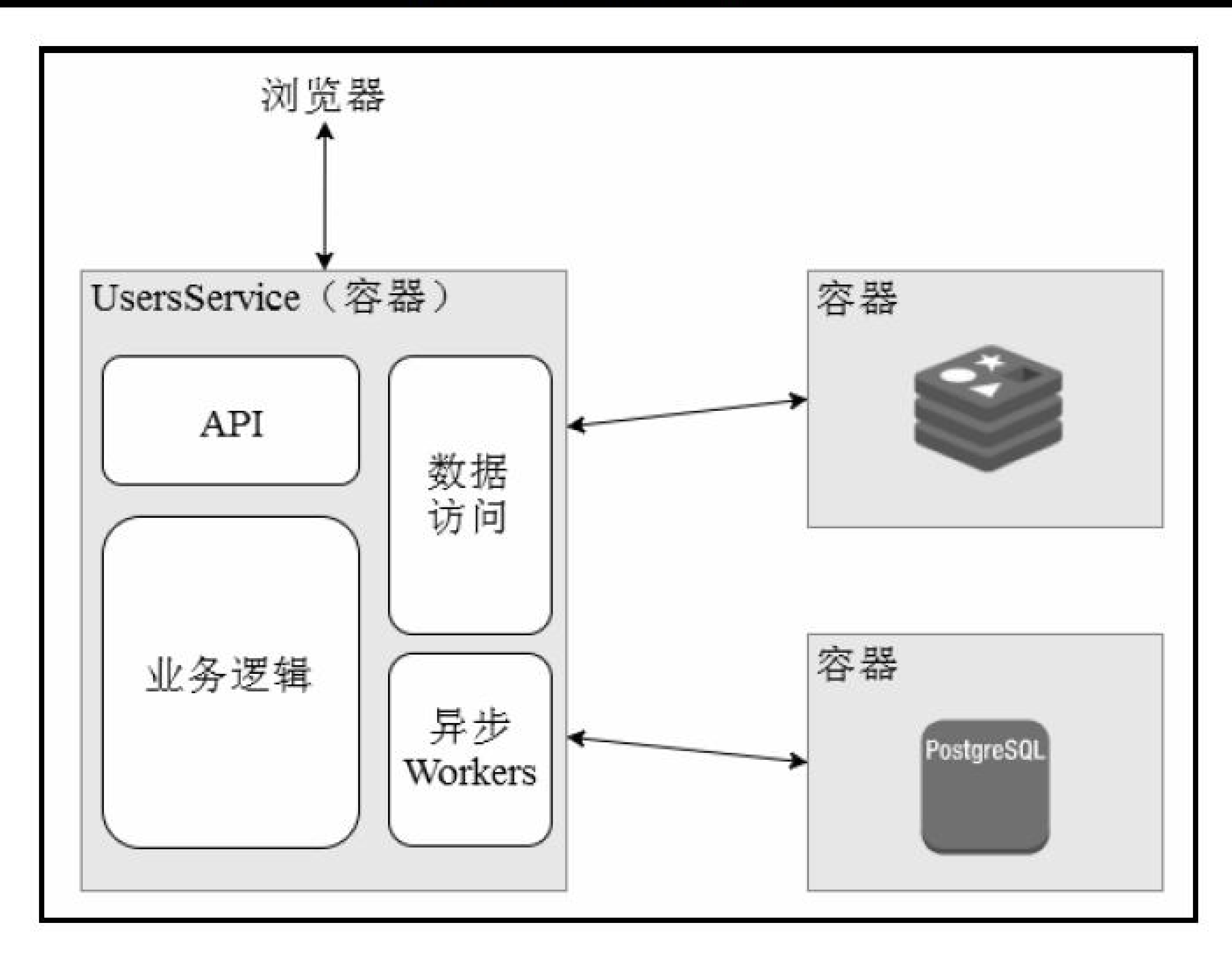


图 4.4

注意：

关于更多控制项，可以从 golang:latest 中手动复制和构建。

LABEL 策略将展示应用程序名称以及版本，如下所示：

```
LABEL Name=userservice Version=0.0.1
```

当采用 RUN 策略时，将创建 Go 的默认工作区目录，这些目录包括 src、pkg 和 bin。对此，可使用“mkdir -p”命令，该命令将创建所有目录，如下所示：

```
RUN mkdir -p /go/src \
  && mkdir -p /go/bin \
  && mkdir -p /go/pkg
```

接下来，将使用 ENV 命令两次，第一次是创建 GOPATH 应用程序；否则，Go 应用程序将无法实现构建过程。随后，再次使用 ENV 命令将 GOPATH 与容器操作系统的 PATH 进行关联，如下所示：

```
ENV GOPATH=/go
ENV PATH=$GOPATH/bin:$PATH
```

再次强调，这里使用了包含“mkdir -p”的 RUN 命令，进而创建应用程序目录，如下所示：

```
# now copy your app to the proper build path
RUN mkdir -p $GOPATH/src/app
```

然后，可使用 ADD 表明容器操作系统中应用程序上的容器位置。类似地，使用 WORKDIR 表示当前工作目录，如下所示：

```
ADD . $GOPATH/src/app
WORKDIR $GOPATH/src/app
```

此处将通过 RUN 运行应用程序的构建过程。相应地，所生成的二进制文件包含了 main 这一名称，并通过 CMD 策略运行，如下所示：

```
RUN go build -o main .
CMD ["/go/src/app/main"]
```

在 Dockerfile 文件的结尾处，我们设置了 3000 端口以访问应用程序端点，如下所示：

```
EXPOSE 3000
```

最终，应用程序完整的 Dockerfile 文件如下所示：

```
# APP Dockerfile
# For more control, you can copy and build manually
FROM golang:latest

LABEL Name=userservice Version=0.0.1

RUN mkdir -p /go/src \
  && mkdir -p /go/bin \
  && mkdir -p /go/pkg
ENV GOPATH=/go
ENV PATH=$GOPATH/bin:$PATH

# now copy your app to the proper build path
RUN mkdir -p $GOPATH/src/app
ADD . $GOPATH/src/app

WORKDIR $GOPATH/src/app
RUN go build -o main .
CMD ["/go/src/app/main"]
EXPOSE 3000
```

在针对当前微服务创建了 Dockerfile 之后，下面将对 PostgreSQL 数据库生成 Dockerfile 文件。

首先可在应用程序中创建名为 db 的新目录。其中，将针对 PostgreSQL 生成 Dockerfile 目录。

该文件相对简单，只需定义基于 FROM 策略下载 PostgreSQL 的存储库。另外，在运行容器的构建过程时，还应执行 create.sql 文件，如下所示：

```
FROM postgres

# run create.sql on init
ADD create.sql /docker-entrypoint-initdb.d
```

下面考察构建容器的 create.sql 文件。

在之前生成的 db 目录中，还可添加除 Dockerfile 之外的另一个文件，即 create.sql 文件。该文件针对测试、开发以及生产环节负责设置数据库，如下所示：

```
CREATE DATABASE users prod;
CREATE DATABASE users dev;
CREATE DATABASE users_test;
```

前述内容定义了两个 Dockerfile 文件，下面返回至 main 目录并创建微服务文件。该文件用于在容器中进行首次编排。

对于编排操作，此处可使用 Docker Compose。因此，在应用程序的 main 目录中，将创建一个 docker-compose.yml 文件。作为 YAML 文件，该文件须遵循相关的标准语法。

在文件开始处，可声明所用的 Docker Compose 版本，并针对服务启用相关语法，如下所示：

```
version: '2.1'
services:
```

这里，声明的第一个服务是 Redis，其中包括容器名称、将要使用的映像、通信端口，以及验证服务是否正在运行的测试行为，如下所示：

```
redis:
    container name: redis
    image: redis
    ports:
    - "6379:6379"
    healthcheck:
      test: exit 0
```

第二个服务则是 PostgreSQL 数据库，该服务包括容器名称、Dockerfile 所处的目录、与当前数据库通信的网关、环境变量，以及验证当前服务是否运行的测试策略。这里需要特别指出的是密码的修改和用户数据库的更改。对应代码如下所示：

```
users-service-db:
    container name: users-service-db
    build: ./db

    ports:
      - 5435:5432 # expose ports - HOST:CONTAINER
    environment:
      - POSTGRES USER=postgres
      - POSTGRES PASSWORD=postgres
    healthcheck:
      test: exit 0
```

第三个服务是当前的微服务。具体来说，UsersService 微服务包括容器名称、Dockerfile 所在的可下载映像，以及该服务的环境变量（取决于访问门和访问链接）。

需要注意的是，当前针对每个数据库使用了连接字符串，并在 db 中的 create.sql 文件中创建，对应代码如下所示：

```
userservice:
    container name: userservice
    image: userservice
    build: .
    environment:
      - APP RD ADDRESS=redis:6379
      - APP RD AUTH=password
      - APP RD DBNAME=0
      - APP SETTINGS=project.config.DevelopmentConfig
      - DATABASE URL=postgres://postgres:postgres@users-service-db:
5432/users prod?sslmode=disable
      -DATABASE DEV URL=postgres://postgres:postgres@users-
service-db:5432/users dev?sslmode=disable
      -DATABASE TEST URL=postgres://postgres:postgres@users-
service-db:5432/users test?sslmode=disable
    depends on:
      users-service-db:
        condition: service healthy
      redis:
        condition: service healthy
    ports:
      - 8080:3000
    links:
      - users-service-db
      - redis
```

最后，Docker-compose.yml 文件如下所示：

```
version: '2.1'

services:
  redis:
    container name: redis
    image: redis
    ports:
     - "6379:6379"
    healthcheck:
      test: exit 0

  users-service-db:
    container name: users-service-db
    build: ./db
    ports:
      - 5435:5432 # expose ports - HOST:CONTAINER
    environment:
      - POSTGRES USER=postgres
      - POSTGRES PASSWORD=postgres
    healthcheck:
      test: exit 0

  userservice:
    container name: userservice
    image: userservice
    build: .
    environment:
      - APP RD ADDRESS=redis:6379
      - APP RD AUTH=password
      - APP RD DBNAME=0
      - APP SETTINGS=project.config.DevelopmentConfig
      - DATABASE URL=postgres://postgres:postgres@users-service-db:
5432/users prod?sslmode=disable
      - DATABASE DEV URL=postgres://postgres:postgres@users-service-db:
5432/users dev?sslmode=disable
      - DATABASE TEST URL=postgres://postgres:postgres@users-service-db:
5432/users test?sslmode=disable
    depends on:
      users-service-db:
        condition: service healthy
      redis:
```

```
        condition: service_healthy
    ports:
      - 8080:3000
    links:
      - users-service-db
      - redis
```

除此之外，还需在当前应用程序中进行适当调整，进而正确地使用容器。

如前所述，在 main.go 文件中，曾利用数据库构成了连接字符串，如下所示：

```
connectionString := fmt.Sprintf(
  "user=%s password=%s dbname=%s sslmode=disable",
  os.Getenv("POSTGRES_USER"),
  os.Getenv("POSTGRES_PASSWORD"),
  os.Getenv("POSTGRES_DB"),
)
```

当在容器中为数据库的连接字符串声明一个环境变量时，这种类型的连接字符串组合并没有什么实际意义。下面将其修改为对环境变量的简单调用，如下所示：

```
connectionString := os.Getenv("DATABASE_DEV_URL")
```

其中，使用到了当前数据库。除此之外，该文件中须调整的内容还包括应用程序服务器运行的端口，如下所示：

```
a.Run(":3000")
```

全部工作均已就绪，下面将使用基于隔离容器的微服务。

2. 使用容器

前述内容已经声明了容器，下面将考察如何使用此类容器。对此，有必要安装 Docker docker-machine 和 docker-compose。

若将 Docker 用作服务，可通过 docker-machine 创建主机，如下所示：

```
$ docker-machine create dev
```

根据生成的主机，下列代码将映射所创建的 docker-machine URL 主机。

```
$ eval "$(docker-machine env dev)"
```

下列代码使用 docker-compose 创建容器，并对其进行初始化。

```
$ docker-compose up -d -build
```

几分钟后，全部容器均处于可用状态。我们只需要知道 Docker 何时运行应用程序即

可。为此，可执行以下命令：

```
$ docker-machine ip dev
```

通过对应的 IP，将端口添加至其结尾处即可开始使用微服务。此时，全部依赖关系均安装完毕且具有更大的灵活性，进而可较好地处理当前应用程序。

4.2 存储分布

在任何应用程序中，数据保存均是一个十分重要的话题，微服务也不例外。利用分布式应用程序，数据的分布将变得更加灵活。

在处理微服务中的存储问题时，存在一些较好的实践方案。显然，CQRS 模式十分有效，但在性能方面仍有所欠缺。对于应用程序的健康问题，数据的区域化和折旧（depreciation）较为有用。

在容器的创建过程中，可以看到，除了应用程序自身的容器之外，在 UsersService 中，还有两个数据存储层的特定容器，并作为数据库本身的缓存。

4.2.1 折旧数据

在科学计算年代，数据分析占有重要地位。删除数据听起来是十分不合理的。是的，这的确有些荒谬。类似地，拥有 100 万数据行的数据库会生成越来越慢的查询操作。

这里的问题是，如果无法从数据库中删除数据，我们应该如何处理才能不影响查询操作？这个问题的答案是数据折旧。数据的折旧包括划分活动数据和非活动数据，并将非活动数据移至与实时应用程序层无关的存储中。

下面考察门票销售这一示例。其中，每天都会有事件被创建和执行，并完成基于事件的门票购买活动。这也是该应用程序的主要例程。

一段时间后，我们将对与事件相关的数据进行审计，或者仅用于数据分析。对于最近的事件、仍处于活动状态的事件，以及已超出 1 年的事件，保存其数据并无意义。

但是，随着时间的推移，将所有数据保存在一个存储设施中，会产生较慢的查询操作，以及更为严重的数据迁移或修改问题。针对于此，数据折旧是一种较好的方法，特别是在实现自动化时。

4.2.2 区域化数据

在应用程序中，通常的做法是将其部署到服务器中，并为应用程序提供访问点——如

果应用程序在全球范围内使用，则重复执行这一处理过程。但通常情况下，服务器的部署一般在距新产品市场最近的地理位置处进行，这一做法完全正确，但问题也会随数据库服务器的地理位置而出现。

查看依据地理位置分布的应用程序是一类常见操作，但需要从几公里之外的服务器处获取数据。因此，在不同的地理位置处，这将带来不同的体验。例如，如果服务器设在美国，那么，美国终端用户的体验要远远好于澳大利亚的用户，其中会涉及物理距离、延迟以及可能出现的数据表丢失问题。针对这一类问题，数据的区域化则是较好的方法。对于新闻门户网站来说，欧洲用户一般仅关注本地的新闻，而不是南非等国家。这意味着，当编辑出版系统发布一个新词时，必须首先按区域存储数据，然后在类似 CQRS 处理过程中对数据进行标准化。稍后，考虑到当前主题的特殊性，对应数据无须进行标准化处理。

基于地理位置划分的数据分布策略涵盖了多样化特征，但不应忽视数据的区域化问题，因为它将直接影响到应用程序的性能。

4.3 隔离——使用生态系统防止故障的出现

本章前述内容讨论了故障的防止措施，以及如何开发具有高可用性和弹性的应用程序。本节将考察相关结构的保护方式。

4.3.1 冗余设计

尽管我们的应用程序仅在单实例中执行，并且由于组件在容器中的分布而变得更加灵活，但仍然会受到系统故障的影响。

冗余是解决上述问题的一种有趣的方法。使用冗余方案时，即使应用程序的一个节点丢失，其他节点仍可以继续响应。

负载平衡器是微服务冗余的一个很好的例子，它使用一种策略来重定向请求。其中，我们可以创建大量节点，如果某个节点出现故障，仍然有另一个节点可以响应。

在图 4.5 中，请求发送至负载平衡器中，并根据开发团队和相关操作制定的某些规则，定向至业务层的实现中。

应用程序的所有节点一般不会出现整体崩溃。如果出现任何不稳定现象，我们仍有时间升至某个新版本；或者简单地识别和修复所遇到的错误。

下面针对当前应用程序使用负载平衡器。对于大型应用程序来说，一种较好的方法

是使用 HAProxy。但在当前示例中，我们令 Nginx 担负起这一职责。

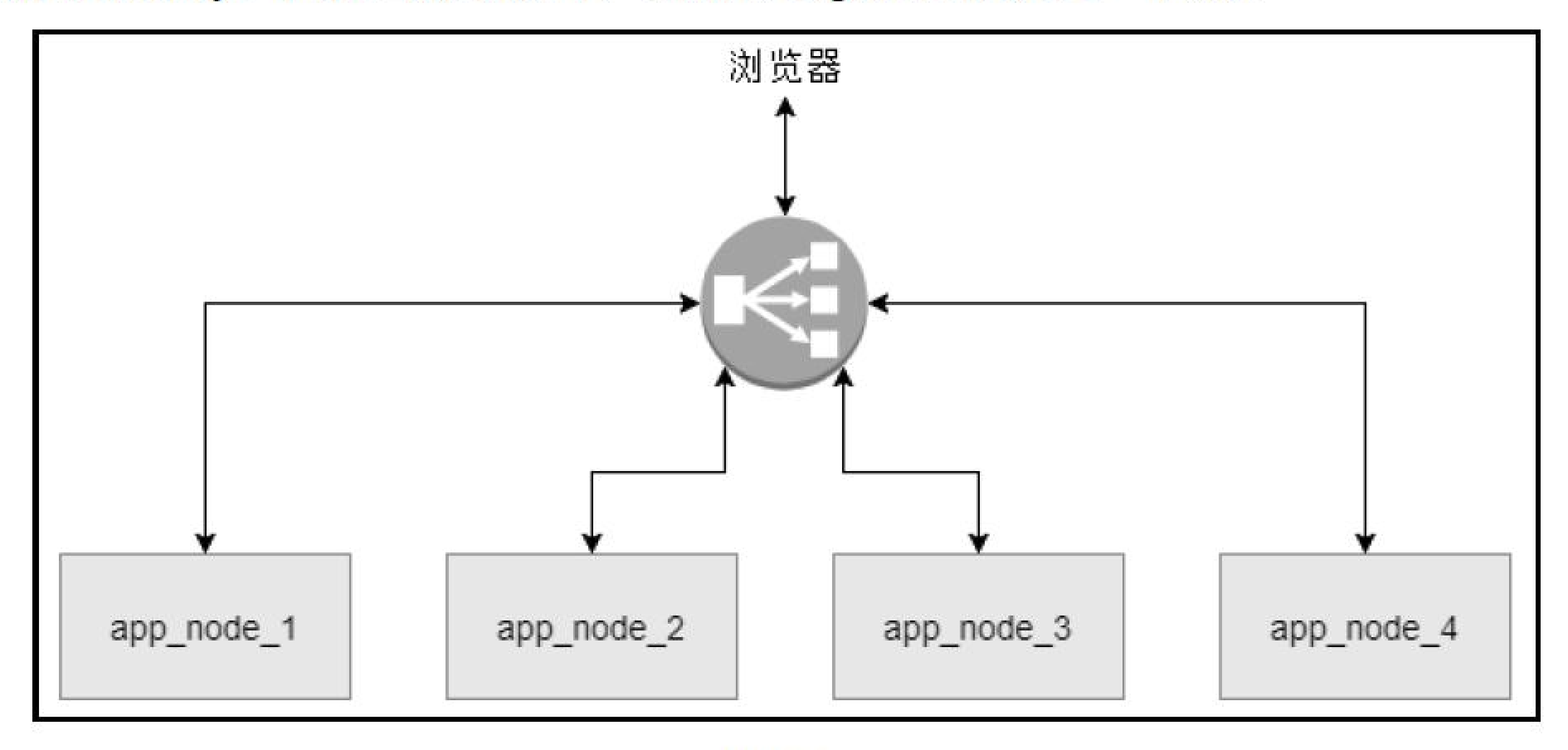


图 4.5

在应用程序的主目录中，为了清晰起见，可创建一个名为 nginx 的目录。在该目录中，还将生成两个新文件，即 Dockerfile 和 nginx。这两个文件已可满足当前应用程序的要求。

下面首先考察 Dockerfile 文件。在该文件中，将通知 Docker 使用哪个映像，以及如何处理 nginx 文件的副本。这里，仍然需要在安装 Nginx 的、操作系统的/etc/nginx 目录中创建该文件，如下所示：

```
FROM nginx
COPY nginx.conf/etc/nginx/nginx.conf
```

在声明了 Dockerfile 之后，将创建 Nginx 配置文件，其中包含了多个 worker，如下所示：

```
worker_processes 4;
```

在此之后，我们将使用事件的配置结果。在这种情况下为每秒 1024 个客户端，如下所示：

```
events { worker_connections 1024; }
```

在初始化设置完毕后，下面将对服务器自身加以考察，并采用 HTTP 策略打开。此处，最重要的事情是设置应用程序节点的 upstream 策略。在 upstream 声明的内容将是在 Nginx proxy_pass 上运行的内容。注意，在 Docker 初始化容器之后，将传递容器的名称以及容器运行的端口，如下所示：

```
upstream user_servers {
    server userservice_userservice_1:3000;
    server userservice_userservice_2:3000;
    server userservice_userservice_3:3000;
    server userservice_userservice_4:3000;
}
```

最终，nginx.conf 文件如下所示：

```
worker processes 4;

events { worker connections 1024; }

http {
    sendfile on;

    upstream user servers {
        server userservice userservice 1:3000;
        server userservice userservice 2:3000;
        server userservice userservice 3:3000;
        server userservice userservice 4:3000;
    }

    server {
        listen 80;

        location / {
            proxy pass http://user servers;
            proxy redirect off;
            proxy set header Host $host;
            proxy set header X-Real-IP $remote addr;
            proxy set header X-Forwarded-For $proxy add x forwarded for;
            proxy set header X-Forwarded-Host $server name;
        }
    }
}
```

随后，需要修改 docker-compose.yml 文件，并创建负载平衡器。首先，通过移除 container_name 语句和端口语句以修改 userservice 声明。之所以删除 container_name 语句，其原因在于，Docker 不会动态地为容器的每个实例创建名称。随后则是删除端口，因为现在需要访问仅由微服务 Nginx 服务器执行的端口，该端口已经设置于 Dockerfile 中。这里只是想告诉读者，端口应指向 Nginx，这些都是已在 nginx.conf 文件中声明的内容，如下所示：

```
userservice:
    image: userservice
    build: ./UsersService
    environment:
      - APP RD ADDRESS=redis:6379
      - APP RD AUTH=password
      - APP RD DBNAME=0
      - APP SETTINGS=project.config.DevelopmentConfig
      - DATABASE URL=postgres://postgres:postgres@users-service-db:
5432/users prod?sslmode=disable
      - DATABASE DEV URL=postgres://postgres:postgres@users-service-db:
5432/users dev?sslmode=disable
      - DATABASE TEST URL=postgres://postgres:postgres@users-service-db:
5432/users test?sslmode=disable
    depends on:
      users-service-db:
        condition: service healthy
      redis:
        condition: service healthy
    links:
      - users-service-db
      - redis
```

接下来将在 docker-compose.yml 文件中声明 Ngnix 服务，对应处理过程类似于 Docker 其他服务的声明。例如，针对容器创建名称、Dockerfile 的构建位置、Ngnix 的访问端口，以及容器的链接，如下所示：

```
proxy:
    container name: userservice loadbalance
    build: ./nginx
    ports:
      - "80:80"
    links:
      - userservice
```

最终，docker-composer.yml 文件包含以下形式：

```
version: '2.1'

services:
  redis:
    container name: redis
    image: redis
```

```
    ports:
     - "6379:6379"
    healthcheck:
      test: exit 0

  users-service-db:
    container name: users-service-db
    build: ./db
    ports:
      - 5435:5432 # expose ports - HOST:CONTAINER
    environment:
      - POSTGRES USER=postgres
      - POSTGRES PASSWORD=postgres
    healthcheck:
      test: exit 0

  userservice:
    image: userservice
    build: ./UsersService
    environment:
      - APP RD ADDRESS=redis:6379
      - APP RD AUTH=password
      - APP RD DBNAME=0
      - APP SETTINGS=project.config.DevelopmentConfig
      - DATABASE URL=postgres://postgres:postgres@users-service-db:
5432/users prod?sslmode=disable
      -DATABASE DEV URL=postgres://postgres:postgres@users-service-db:
5432/users dev?sslmode=disable
      -DATABASE TEST URL=postgres://postgres:postgres@users-service-db:
5432/users test?sslmode=disable
    depends on:
      users-service-db:
        condition: service healthy
      redis:
        condition: service healthy
    links:
      - users-service-db
      - redis

  proxy:
    container name: userservice loadbalance
    build: ./nginx
    ports:
```

```
    - "80:80"
  links:
    - userservice
```

这种方法非常简单，但是也向微服务中引入了某种程度的冗余内容，其想法是使用 Apache Mesos、Swarm 和 Kubernetes 等工具，因而会更多地关注弹性方面的内容。

4.3.2　临界分区

在理解我们正在开发的应用程序时，重要的一点是了解应用程序最常使用的访问点，而此类访问点往往难以发现。

假设我们正与某个在线销售系统协同工作，在该应用程序类型中包含了多种组件，但显然，最为重要的组件是完成产品销售部分的组件。然而，全部销售流程与业务逻辑之间处于耦合状态，这种耦合源自销售界面直至付款。

上述在线商店似乎工作良好。只要向服务器群添加更多的机器，系统的所有压力都可以通过水平可扩展性得到解决。这里的问题是，针对当前应用程序类型，水平扩展并非绝对可行。其中，许多资源处于共享状态并可立即获得。对于一些难以令人满意或者是无关紧要的结果，过程中的延迟将会造成很高的代价。再次强调，对于应用程序来说，“成功”一词往往意味着严峻的问题。例如，网上商店在“黑色星期五”时即会面临此类问题。

在几分钟内，可以查看到大量的点击量，但是销售额却未见明显增长，这绝非是正常现象。几分钟之后，就会发现问题所在。在这种情况下，主要问题来自处于耦合状态的购买流。其间，在线商店的访问量巨大，以至于阻塞了后续的购买流程。也就是说，搜索价格的用户屏蔽了广告产品的实际购买者。如果这种偏差十分严重，那么，应采取相关措施处理此类问题。

这种在实时交互性应用程序中的故障是非常常见的。通常，点击量一般大于完成整个交互性流程的用户数量。此类场景适用于任何类型的实时应用程序，无论是在线商店、游戏还是政府的税务申报系统。所有这些应用程序均包含相同的特征：高季节性访问以及低转换率。产品的转换数量一般不会受到影响，即使该数量低于点击量。

当处理各类问题时，微服务架构通常十分有效。当采用 DDD 对每个域内的边界上下文进行设置，以及执行关键评估时，应用程序每个部分可选取更加智能的架构。

在图 4.6 中，采用临界划分这一概念可使处理操作升至当前最为需要的程序段。据此，软件的其他部分则可避免外部大量点击操作造成的压力。

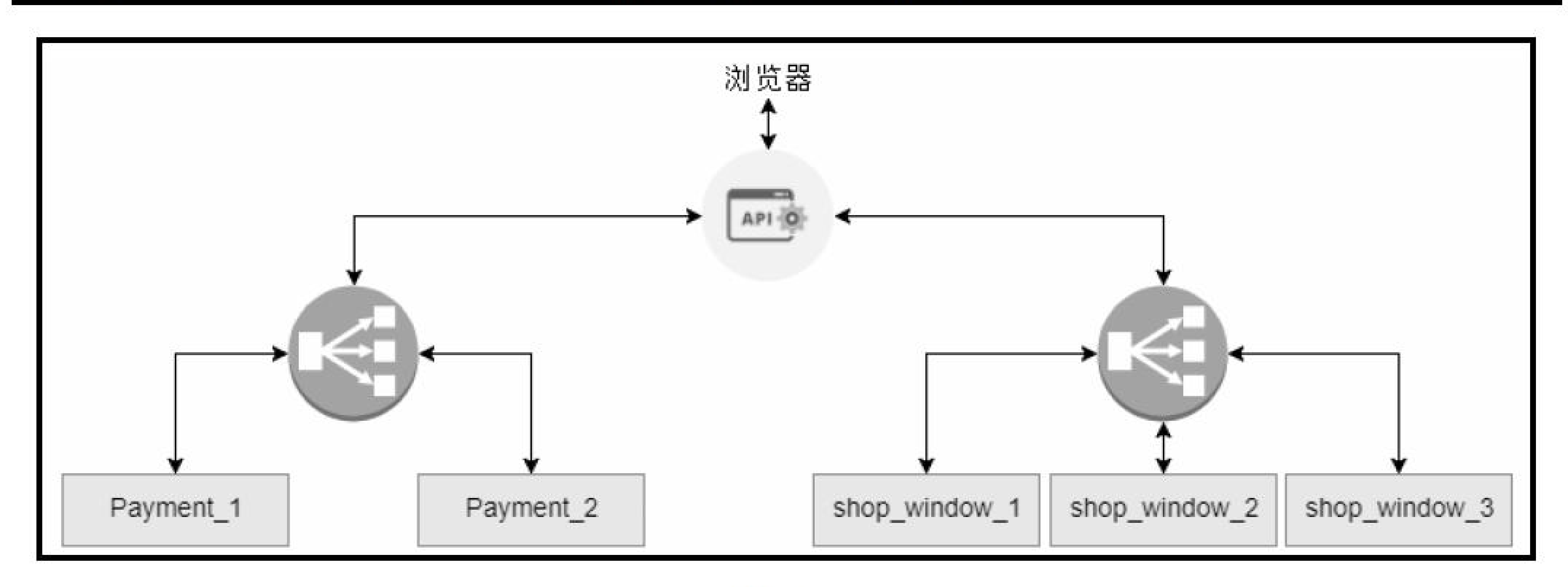


图 4.6

4.3.3 隔离设计

我们已经看到，通过临界状态分离应用程序包含了诸多优点。然而，一个常见的误解是组件的复用。对此，一种较差的资源共享示例是数据库的复用。无论负载平衡器、应用程序线程级别或者域划分方式如何优化，如果应用程序仅依赖于单一物理组件，那么，系统崩溃迟早会到来。

图 4.7 中的设计方式即显得较为草率。其中，应用程序的全部组件（暂不考虑相关域）均依赖于同一物理组件，在该示例中则是数据库。这一类错误也常出现于缓存和消息代理中。

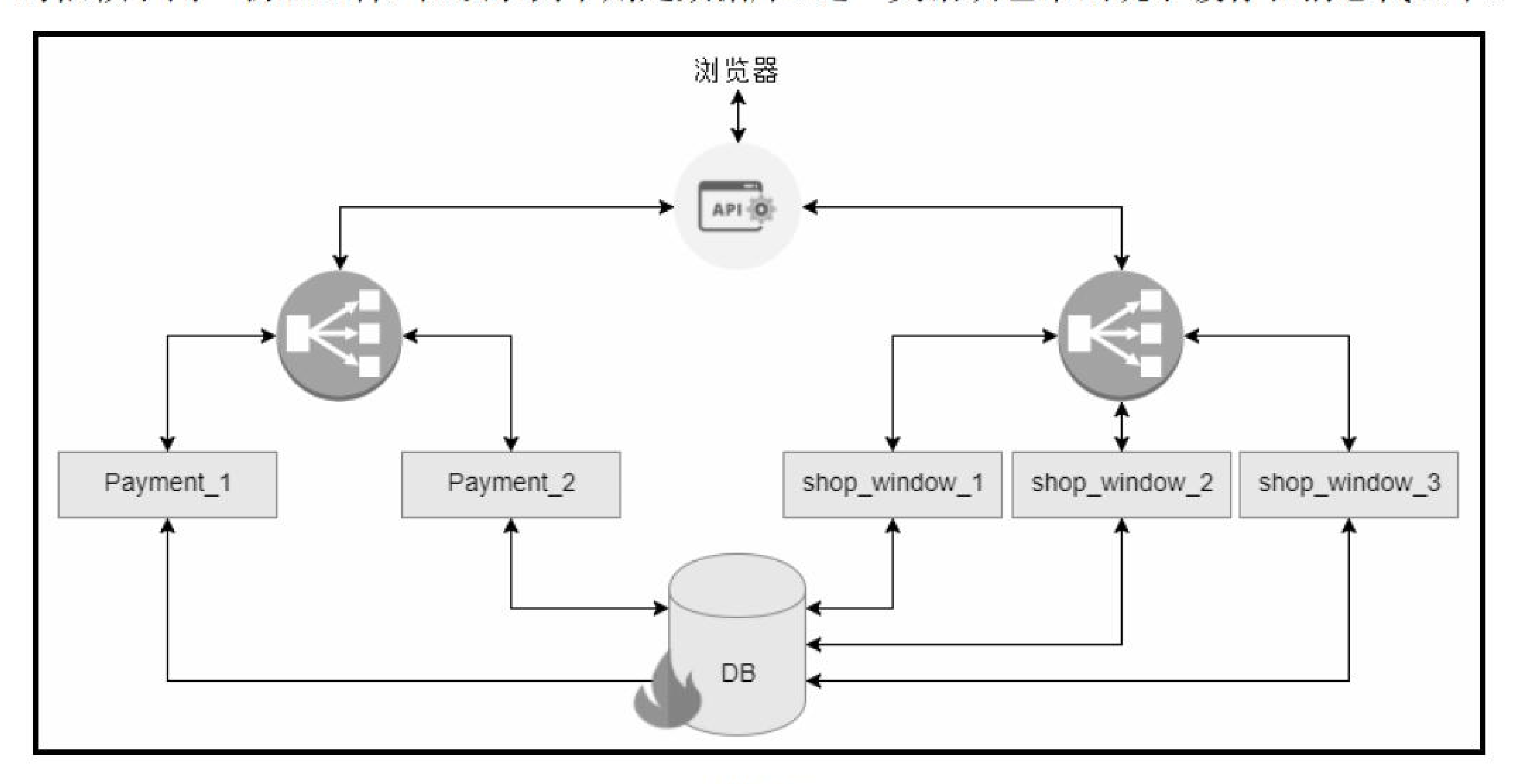


图 4.7

需要注意的是，无论是否缓存，数据也是微服务域中的一部分内容。物理组件应尽可能地处于独立状态，如图 4.8 所示。

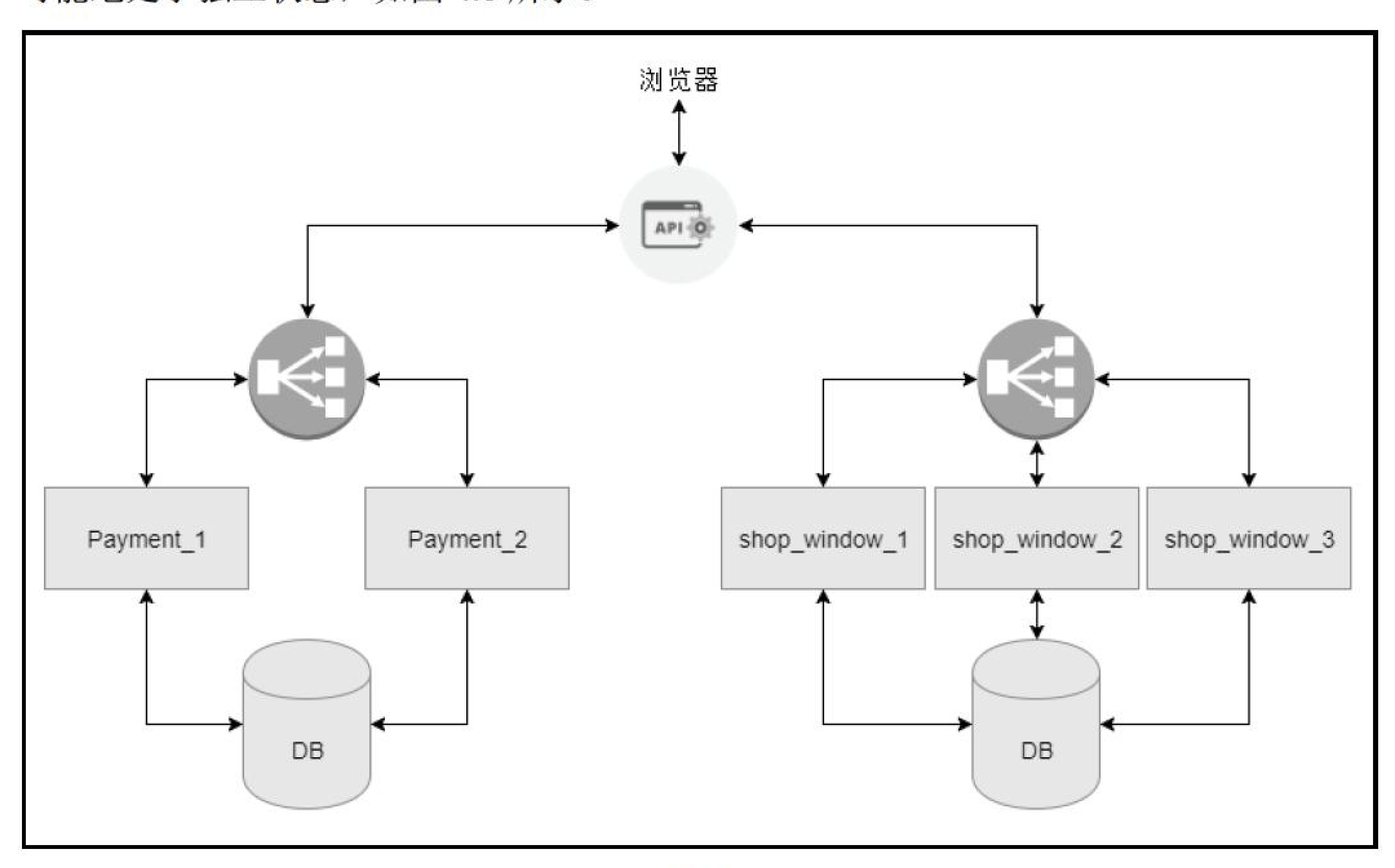


图 4.8

图 4.8 中所示的物理组件方案则更为常见，进而可简化迁移和数据库的分片机制。

4.3.4　快速故障

本节将考察一个在线商店示例。如前所述，就可用性来说，临界划分并遵循 DDD 边界包含诸多优点。除此之外，前述内容还讨论了软件与物理组件依赖关系中蕴含的风险。相应地，其中的大部分内容可消除系统架构中的故障或风险。有时，问题并不取决于我们，外部服务也会在生态系统中使用微服务。

对于在线商店的支付流程，微服务则是购物流程中的最后一个步骤，并负责完成产品的付款操作。付款过程必须通过信用卡网关进行通信，如果这些网关中的任何一个出现中断，鉴于外部服务的依赖关系，微服务可能会出现故障。

在图 4.9 中，微服务支付过程尝试构建与计费系统间的通信，但其中存在某些问题。

这里，我们不能简单地等待连接被恢复；相反，此处应产生快速故障并及时进行下

一步处理。对于此类问题，断路器是一种较好的方法。据此，可以设置超时或连接故障策略，进而提供其他形式的付款方式，或者以友好的方式向用户提供故障信息，从而避免了由于资源耗尽而导致的系统故障。

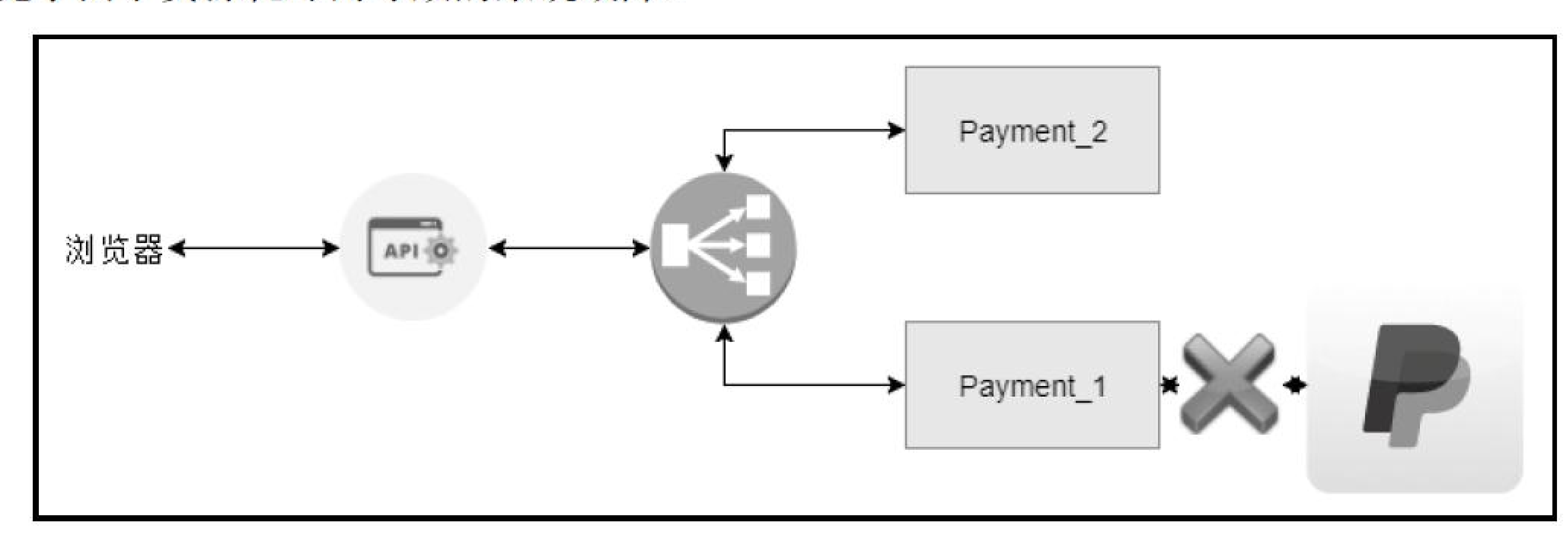


图 4.9

4.4 断　路　器

断路器是当出现过载或短路时自动关闭的操作开关。与电路保险丝类似，断路器的目的是实现快速故障和保护电力装置。对于微服务来说，断路器可保护应用程序的整体完整性。

假设微服务的运行速度变得十分缓慢。其中，各方请求不断出现并开始排队。在某种程度上，这可视为一种间接损害。特别是对于依赖于与其他微服务通信的微服务，此时需要使用断路器。

断路器的概念较为简单，其中仅包含了两种状态，如下所示。

- 开：释放对外部依赖关系的调用。
- 关：即刻放弃当前调用，并执行先前配置的操作。

在实际操作过程中，断路器自身将置于调用中，而不是微服务直接访问外部依赖项。对于基于预定义参数产生的任何故障，例如超时，断路器将中断与故障依赖项间的通信。当然，微服务方面也可以采取一些措施。断路器的行为如图 4.10 所示。

一些框架可以帮助实现断路器。目前，较为突出的框架是由 Netflix 开发团队创建的 Hystrix。Hystrix 最初是在 Java 中开发的，但是已经有其他编程语言实现了该算法，如 Go 语言。

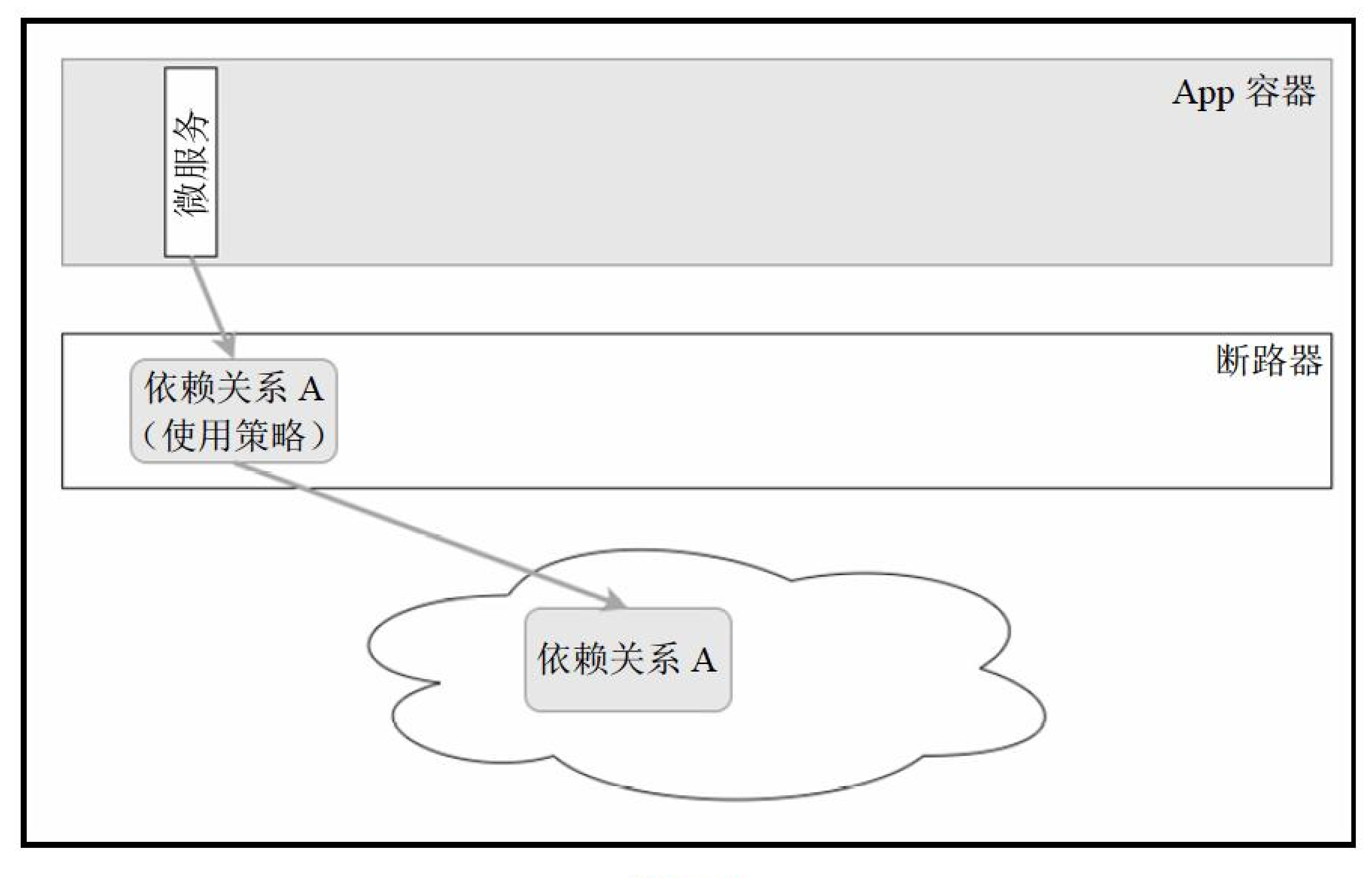


图 4.10

4.5 本 章 小 结

本章讨论了一些较好的实践方案，以使微服务更具弹性和容错能力。另外，本章还介绍了应用程序中容器的使用方式，并使用 Nginx 开发 UsersService 集群。

最后，本章还讨论了模式方面的内容，并可应用于任何技术上。第 5 章将继续考察新闻门户网站项目。

第 5 章　共享数据微服务设计模式

本章将介绍更多关于微服务架构方面的内容，并通过相关示例探讨了多种模式。此外，还介绍了开发人员几乎每天都会遇到的一些重要概念。所有的新概念都具有一定的指导意义，目的是使用微服务架构为开发提供更多的便利、安全和速度。

但是，大多数概念和技术都关注于新项目并使用微服务架构。在某些情况下，一些遗留项目中的内容往往与另一个架构模式相背离。少数时候，还存在一些架构转换案例。

通常，由于缺乏文档以及最终目标，针对微服务的迁移行为将变得十分复杂。本章主要讨论遗留项目和微服务之间的转换行为。

本章主要涉及以下内容：

- 分解单体应用程序。
- 开发新的微服务。
- 数据编排。
- 对响应进行整合。
- 微服务通信。
- 反模式。
- 一些最佳实践方案。

5.1　理 解 模 式

任何遗留的新设计过程都被称为绿色项目或绿地（greenfield）应用。相比较而言，项目中已经存在的部分通常称为棕地（brown）项目。显然，相比于遗留项目，在绿色项目中应用领域驱动设计（DDD）和模式将更加简单。

当谈及微服务时，共享数据模式则是一种有争议的模式。如果将其应用至绿色项目中，共享数据模式常视作一种反模式。但是，对于处于过渡阶段的遗留应用程序来说，这种模式应该被视为一种临时模式。

该模式背后的概念是针对数据存储使用相同的物理结构，当对数据结构产生某种疑虑时，或者是微服务间通信层未经良好定义时，即可使用这种模式。

图 5.1 显示了共享数据模式的工作方式。

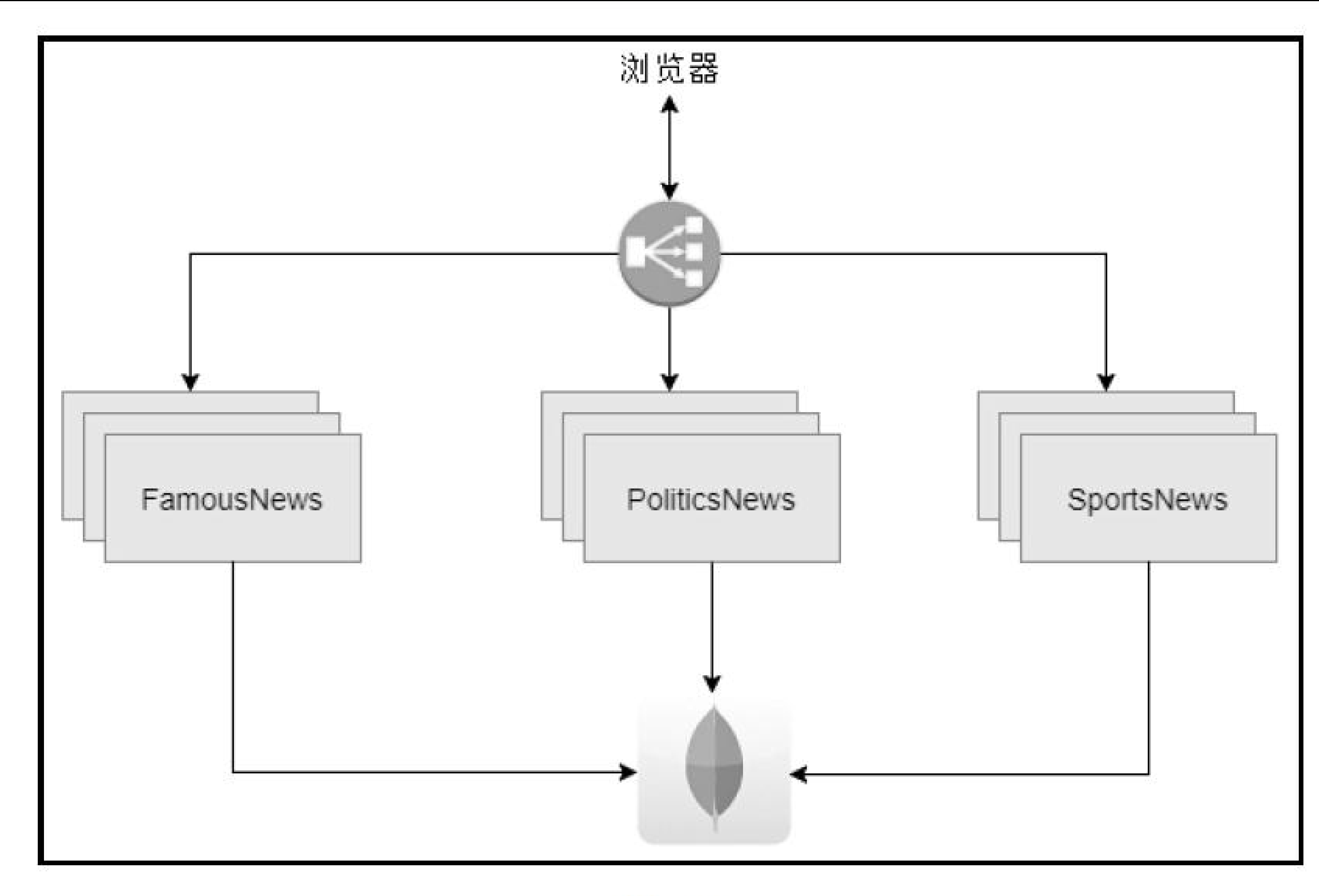


图 5.1

5.2　将单体应用程序划分为微服务

需要注意的是，在过渡阶段的初期，一切似乎毫无头绪，微服务迁移往往会令人认为是一个巨大的错误。

成功的迁移工作往往伴随着挫折与压力，因此，在项目初期即制订规划是非常重要的，其中包括清晰的目标和期限。

显然，按期交付是一项最大的挑战，尤其是面临未知领域时。这里，规定最后的期限十分重要，它可以帮助我们了解项目处于哪一个阶段。

虽然一些步骤看似不必要或者不相关，但是，如果缺少这些步骤，任何用于微服务的迁移项目都会遭遇严重失败。下面将对执行序列中所有步骤加以讨论，相关步骤如下所示：

（1）定义优先级。

（2）设置期限。

（3）定义应用程序域。

（4）试验操作。

（5）制定标准。

（6）构建原型。

（7）发送至产品中。

5.2.1　定义优先级

许多公司都希望拥有最好的开发生态环境，以及最快的实现速度，殊不知这一切都需要大量的投资。对于软件架构的迁移，投资不仅指财力，同时还会涉及与时间相关的一切事物。

对于公司来说，如果微服务开发无法定义为某种优先级，那么，建议最好不要启动该项目。迁移行为既复杂又困难，在开始阶段更像是一场灾难。如果迁移的优先级未经良好定义，项目往往会中途终止。因此，与保持单体系统相比，公司将面临更加糟糕的状况。最坏的情况是，当持有一个混合应用程序时，一部分内容位于单一系统中，而另一部分则位于微服务中。随着时间的推移，在应用程序中代表业务的地方可能会出现歧义，并且维护这个应用程序会变得更加困难。

5.2.2　设置期限

为迁移过程设置最后期限是很重要的。对于项目健康和进展程度，最后期限可视作是一种标准。显然，随着复杂度的不断累积，最后期限也可进行适当调整。

最后期限应与微服务相关，可能包括堆栈定义的最后期限、域定义、早期开发的日期，以及对性能测试运行时间方面的预测。

5.2.3　定义应用程序域

第 1 章曾讨论了 DDD、定义以及对业务层的限制条件。正是在这一点上，我们所有关于 DDD 的知识都得到了实际应用。

如果缺乏定义良好的域，迁移过程将十分耗时且极易出错，这也使得重新设计过程会涉及大量烦琐的步骤。

5.2.4　试验操作

在再次应用任何技术或模式之前，应尝试对其进行试验。在某些情况下，会存一些模拟场景中生成的文档，但是这些场景可能被过度模拟，或者无法反映现实状况。

所有的技术开发结果都应在经过严格的试验之后方可投入使用，同时还应记录连续的异步场景，其中涉及加载进度以及大量的即时加载。因此，试验过程不可或缺，最后

还应提交确定的试验结果。

5.2.5　制定标准

在试验结束后，还应对各方标准加以定义。默认标准指的是团队、文档、语言以及流程间的通信模式，而不是注释或代码的架构。如果标准缺乏应有的清晰度，一段时间后，原有计划将无法实现。

5.2.6　构建原型

我们可以对诸多事物加以思考、编写和定义，但若缺乏实际的测试过程，真实性将无从谈起。当然，冒险涉入一个新的生态系统蕴含了很大的风险，所以最好创建一个原型对概念、试验结果和模式的有效性进行验证。

创建原型的一个较好的方法是，选择最小规模和最简单的领域，以及对于整体应用程序来说，可以有效地规避风险的领域。

5.2.7　发送产品

尽管可以执行多项测试，且原型可能已获得成功，但均无法与产品中的验证相提并论。这听起来令人绝望或疯狂，但对于微服务架构来说，这确实是一个巨大的优势。

向产品发送微服务可通过渐进方式进行并且是可控的。我们的想法是发送一些不成熟的产品，让客户对其进行验证。事实上，这将有助于对发布过程加以控制，并获取实际的度量结果。

5.2.8　开发新的微服务

截至目前，前述内容已经实现了 UsersService，该微服务负责处理用户数据。下面开始讨论负责操控数据的微服务，其中包括：

- FamousNewsService。
- PoliticsNewsService。
- SportsNewsService。

当前域已经在前面的章节中加以定义。下面将编写这些微服务的代码，并从 FamousNewsService 开始。

1. 编写微服务配置文件

针对每种开发环境，config.py 文件包含了相关的设置项。该文件被划分为多个类，

其中包含了每种环境定义。

相应地，第一个类是 BaseConfig。该类仅用作其他类的继承结果，如下所示：

```
class BaseConfig:
    """Base configuration"""
    DEBUG = False
    TESTING = False
    MONGODB_SETTINGS = {}
```

在 BaseConfig 声明完毕后，下面将声明其他类，进而操控每种环境的配置过程。此处，当前环境包括 Development、Test 以及 Production，分别表示每种环境。对应的类则定义为 DevelopmentConfig、TestingConfig 和 ProductionConfig。

上述类具有等同的结构形式，唯一的不同之处是与数据库的连接字符串，如下所示：

```
class DevelopmentConfig(BaseConfig):
    """Development configuration"""
    DEBUG = True
    MONGODB_SETTINGS = {
        'db': 'famous_dev',
        'host': '{}{}'.format(
            os.environ.get('DATABASE_HOST'),
            'famous_dev',
        ),
    }
class TestingConfig(BaseConfig):
    """Testing configuration"""
    DEBUG = True
    TESTING = True
    MONGODB_SETTINGS = {
        'db': 'famous_test',
        'host': '{}{}'.format(
            os.environ.get('DATABASE_HOST'),
            'famous_test',
        ),
    }

class ProductionConfig(BaseConfig):
    """Production configuration"""
    DEBUG = False
    MONGODB_SETTINGS = {
        'db': 'famous',
        'host': '{}{}'.format(
```

```
            os.environ.get('DATABASE HOST'),
            'famous',
        ),
    }
```

2. 创建模型

在开始阶段，models.py 中仅包含了数据库中的一个实体，即 News。对应结构也较为简单，并可在后续过程中增加内容。这可视作一种较好的建议：从简单开始，随后进行测试并编写有效的代码。这里，News 类体现了应用程序中域的中心内容。

下面首先声明导入语句，如下所示：

```
import datetime
from flask_mongoengine import MongoEngine
```

随后声明 MongoEngine 实例，它是用来为源自 MongoDB 的数据提供对应结构的工具，如下所示：

```
db = MongoEngine()
```

下面将声明 News 类，其中定义了与实体表达匹配的字段。该类较为简单，对应字段完全具有自解释性，如下所示：

```
class News(db.Document):
    title = db.StringField(required=True, max length=200)
    content = db.StringField(required=True)
    author = db.StringField(required=True, max length=50)
    created at = db.DateTimeField(default=datetime.datetime.now)
    published at = db.DateTimeField()
    news type = db.StringField(default="famous")
    tags = db.ListField(db.StringField(max_length=50))
```

3. 显示微服务数据

利用所声明的模型，即可实现 views.py 文件。如前所述，对于 News 类，我们将使用 flask 作为微服务框架。

首先针对 views.py 文件称作声明所需的导入语句，如下所示：

```
import datetime
import mongoengine
from flask import Blueprint, jsonify, request
from models import News
```

接下来，将从 flask 中实例化 Blueprint 类。该工具需要使用到若干装饰器，并针对应用程序声明访问路由，如下所示：

```
famous_news = Blueprint('famous_news', __name__)
```

这里，所声明的第一个路由负责通过 ID 提供 News——该处理过程较为简单。首先，使用基于 Blueprint 的变量 famous_news 作为装饰器来规定访问路由。注意，我们在装饰器中指出，只接受使用 HTTP GET 谓词的请求。除此之外，还可指定所支持的 ID 类型，此处为字符串——MongoDB ID 表示为唯一的哈希值，如下所示：

```
@famous_news.route('/famous/news/<string:news_id>', methods=['GET'])
```

此后，将声明作为视图的函数名。在该函数内，dict 作为模板以创建视图所发送的响应，如下所示：

```
def get single news(news id):
    """Get single user details"""
    response object = {
        'status': 'fail',
        'message': 'User does not exist'
    }
```

因此，利用成功响应的模板，需要在数据库中通过 ID 进行搜索。该处理过程封装在 try...except 块中。如果传递了无效或并不存在的 ID，MongoEngine 将抛出 DoesNotExist 异常。最后，可通过 jsonify 以及适当的 HTTP 代码将输出格式化为 JSON 结果，如下所示：

```
try:
    news = News.objects.get(id=news id)
    response object = {
        'status': 'success',
        'data': news,
    }
    return jsonify(response object), 200
except mongoengine.DoesNotExist:
    return jsonify(response_object), 404
```

视图中的第一个函数包含下列形式：

```
@famous news.route('/famous/news/<string:news id>', methods=['GET'])
def get single news(news id):
    """Get single user details"""
    response object = {
        'status': 'fail',
```

```
        'message': 'User does not exist'
    }
    try:
        news = News.objects.get(id=news id)
        response object = {
            'status': 'success',
            'data': news,
        }
        return jsonify(response object), 200
    except mongoengine.DoesNotExist:
        return jsonify(response_object), 404
```

所有视图均包含了类似的结果，这里仅需修改每个函数的内部代码，以体现视图的执行内容。

下一个视图负责将全部消息以页面形式加以显示。再次强调，视图的创建过程基本一致。首先，可访问当前路由，然后在数据库中搜索应答模板的成分，接着使用 jsonify 查询应答本身，如下所示：

```
@famous news.route('/famous/news/<int:num page>/<int:limit>',
methods=['GET'])
def get all news(num page, limit):
    """Get all users"""
    news = News.objects.paginate(page=num page, per page=limit)
    response object = {
        'status': 'success',
        'data': news.items,
    }
    return jsonify(response_object), 200
```

下一个函数 views.py 的结构并无太多变化。作为 News 创建视图，只是在该函数内部稍作调整。

该函数也包含了访问路由和名称，其不同之处在于，此处将捕捉请求，并转换 dict 接收的 JSON 结果。如果信息无效，相关 HTTP 代码将返回错误消息。对应代码如下所示：

```
if not post data:
    response object = {
        'status': 'fail',
        'message': 'Invalid payload.'
    }
    return jsonify(response_object), 400
```

经验证后，将利用源自请求中的数据实例化并保存 News，如下所示：

```
news = News(
   title=post data['title'],
   content=post data['content'],
   author=post data['author'],
   tags=post data['tags'],
).save()
```

最后，当前视图具有如下形式，其中包含了持久化层和相应的 HTTP 响应：

```
@famous news.route('/famous/news', methods=['POST'])
def add news():
   post data = request.get json()
   if not post data:
      response object = {
         'status': 'fail',
         'message': 'Invalid payload.'
      }
      return jsonify(response object), 400
   news = News(
      title=post data['title'],
      content=post data['content'],
      author=post data['author'],
      tags=post data['tags'],
   ).save()
   response object = {
      'status': 'success',
      'news': news,
   }
   return jsonify(response_object), 201
```

在当前域中，我们创建了 News，但并未即刻发布。针对该项任务，views.py 文件中定义了另一个函数。

此端点在路由中接收发布的新闻 ID，在数据库中执行相同的搜索，对新闻发布日期进行更新，并返回相应的 HTTP 响应。截至目前，唯一不同之处是更新相关的 News，如下所示：

```
@famous news.route('/famous/news/<string:news id>/publish/',
methods=['GET'])
def publish news(news id):
   try:
      news = News.objects.get(id=news id)
      news.update(published at=datetime.datetime.now)
      news.reload()
```

```
        response object = {
            'status': 'success',
            'news': news,
        }
        return jsonify(response object), 200
    except mongoengine.DoesNotExist:
        return jsonify(response_object), 404
```

显然，News 中的更新行为与发布日期无关。由于人们可能会对 News 的内容产生误解，因而需要对其进行编辑操作。该过程等价于发布日期的更新操作，只是某些信息字段发生了变化，如下所示：

```
@famous news.route('/famous/news', methods=['PUT'])
def update news():
    try:
        post data = request.get json()
        news = News.objects.get(id=post data['news id'])
        news.update(
            title=post data.get('title', news.title),
            content=post data.get('content', news.content),
            author=post data.get('author', news.author),
            tags=post data.get('tags', news.tags),
        )
        news.reload()
        response object = {
            'status': 'success',
            'news': news,
        }
        return jsonify(response object), 200
    except mongoengine.DoesNotExist:
        return jsonify(response_object), 404
```

视图中的最后一个函数负责 News 的删除操作，具体过程与其他视图基本相同，唯一的差别在于数据库中的删除操作，如下所示：

```
@famous news.route('/famous/news/<string:news id>', methods=['DELETE'])
def delete news(news id):
    News.objects(id=news_id).delete()
    response object = {
        'status': 'success',
        'news id': news id,
    }
    return jsonify(response_object), 200
```

最终，完整的 views.py 具有如下格式：

```
import datetime
import mongoengine

from flask import Blueprint, jsonify, request
from models import News
famous news = Blueprint('famous news',   name  )
@famous news.route('/famous/news/<string:news id>', methods=['GET'])
def get single news(news id):
    """Get single user details"""
    response object = {
        'status': 'fail',
        'message': 'User does not exist'
    }
    try:
        news = News.objects.get(id=news id)
        response object = {
            'status': 'success',
            'data': news,
        }
        return jsonify(response object), 200
    except mongoengine.DoesNotExist:
        return jsonify(response object), 404
@famous news.route('/famous/news/<int:num page>/<int:limit>',
methods=['GET'])
def get all news(num page, limit):
    """Get all users"""
    news = News.objects.paginate(page=num page, per page=limit)
    response object = {
        'status': 'success',
        'data': news.items,
    }
    return jsonify(response object), 200

@famous news.route('/famous/news', methods=['POST'])
def add news():
    post data = request.get json()
    if not post data:
        response object = {
            'status': 'fail',
            'message': 'Invalid payload.'
        }
```

```
        return jsonify(response object), 400
    news = News(
        title=post data['title'],
        content=post data['content'],
        author=post data['author'],
        tags=post data['tags'],
    ).save()
    response object = {
        'status': 'success',
        'news': news,
    }
    return jsonify(response object), 201

@famous news.route('/famous/news/<string:news id>/publish/',
methods=['GET'])
def publish news(news id):
    try:
        news = News.objects.get(id=news id)
        news.update(published at=datetime.datetime.now)
        news.reload()
        response object = {
            'status': 'success',
            'news': news,
        }
        return jsonify(response object), 200
    except mongoengine.DoesNotExist:
        return jsonify(response object), 404


@famous news.route('/famous/news', methods=['PUT'])
def update news():
    try:
        post data = request.get json()
        news = News.objects.get(id=post data['news id'])
        news.update(
            title=post data.get('title', news.title),
            content=post data.get('content', news.content),
            author=post data.get('author', news.author),
            tags=post data.get('tags', news.tags),
        )
        news.reload()
        response_object = {
```

```
            'status': 'success',
            'news': news,
        }
        return jsonify(response object), 200
    except mongoengine.DoesNotExist:
        return jsonify(response object), 404

@famous news.route('/famous/news/<string:news id>', methods=['DELETE'])
def delete news(news id):
    News.objects(id=news id).delete()
    response object = {
        'status': 'success',
        'news id': news id,
    }
    return jsonify(response_object), 200
```

4. 运行应用程序

app.py 文件负责运行应用程序。对于该文件，首先需要声明必要的导入语句。需要注意的是，除了 Flask 之外，还需要导入 Blueprint（包含全部路由）以及 MongoEngine 实例（包含了实体声明），如下所示：

```
import os
from flask import Flask
from views import famous_newsfrom models import db
```

下面讨论 Flask 实例化操作，并传递环境设置、数据库实例以及视图路由实例，如下所示：

```
# instantiate the app
app = Flask(   name   )

# set config
app settings = os.getenv('APP SETTINGS')
app.config.from object(app settings)

db.init app(app)

# register blueprints
app.register_blueprint(famous_news)
```

在文件结尾处，还定义了一个简单的语句，并在 5000 端口上执行 Flask，如下所示：

```
if   name   == '  main  ':
    app.run(host='0.0.0.0', port=5000)
```

app.py 文件包含了以下格式：

```
import os
from flask import Flask

from views import famous news
from models import db

# instantiate the app
app = Flask(  name  )

# set config
app settings = os.getenv('APP SETTINGS')
app.config.from object(app settings)

db.init app(app)

# register blueprints
app.register_blueprint(famous_news)

if   name   == '  main  ':
    app.run(host='0.0.0.0', port=5000)
```

5．创建 Dockerfile

之前的应用程序均在容器中进行开发，当前程序也不例外。下面声明微服务 Dockerfile。其中，将使用到 Python 容器，这也是微服务中常见的编程语言。

在该文件中，需要注意两点内容。首先是 requirements.txt 文件，其中包含了所有的项目依赖关系。第二点需要注意的地方是端口 5000，这也是应用程序服务器运行之处。对应代码如下所示：

```
FROM python:3.6.1
COPY . /app
WORKDIR /app
RUN pip install -r requirements.txt
ENTRYPOINT ["python"]
CMD ["app.py"]
EXPOSE 5000
```

6. 基于 requirements.txt 的依赖关系

requirements.txt 文件包含了项目的依赖关系。读者不必担心该文件中的所有依赖项——大多数为依赖关系的依赖项。需要注意的是，在使用该应用程序之前，需要安装相关库，如下所示：

```
appnope==0.1.0
blinker==1.4
click==6.7
decorator==4.1.2
Flask==0.12.2
flask-mongoengine==0.9.3
Flask-Script==2.0.6
Flask-WTF==0.14.2
ipdb==0.10.3
ipython==6.2.1
ipython-genutils==0.2.0
itsdangerous==0.24
jedi==0.11.0
Jinja2==2.9.6
MarkupSafe==1.0
mongoengine==0.14.3
parso==0.1.0
pexpect==4.2.1
pickleshare==0.7.4
prompt-toolkit==1.0.15
ptyprocess==0.5.2
Pygments==2.2.0
pymongo==3.5.1
simplegeneric==0.8.1
six==1.11.0
traitlets==4.3.2
wcwidth==0.1.7
Werkzeug==0.12.2
WTForms==2.1
Flask-Testing==0.6.2
```

就代码而言，上述 3 个 News 微服务彼此相等，仅是路由和设置等内部声明中涵盖了一些变化。

考察下列路由：

```
@famous_news.route('/famous/news/<string:news_id>', methods=['GET'])
@politics_news.route('/politics/news/<string:news_id>', methods=['GET'])
@sports_news.route('/sports/news/<string:news_id>', methods=['GET'])
```

接下来，考察下列配置内容：

```
MONGODB SETTINGS = {
        'db': 'famous_test',
...

MONGODB SETTINGS = {
        'db': 'politics_test',
...

MONGODB SETTINGS = {
        'db': 'sports_test',
...
```

实际上，微服务具有相同的结构，但每个微服务的演变和开发过程则是完全独立的。

5.3　数 据 编 排

在共享数据模式中，无须进行数据编排，其原因在于，微服务对于数据存储使用相同的物理组件。而在相关的 News 微服务中，这可在应用程序的 docker-compose.yml 文件中进行查看。

在归档文件中，我们添加了 MongoDB 作为数据库，对应的声明过程十分简单，只需为容器命名即可，并指定 MongoDB 的位置，同时为 MongoDB 的功能提供执行命令，如下所示：

```
mongo:
   image: mongo:latest
   container name: "mongodb"
   ports:
      - 27017:27017
   command: mongod --smallfiles --logpath=/dev/null # --quiet
```

下面开始添加之前创建的微服务，这 3 项服务有相同的模式。这里的重点内容是环境变量。变量 APP_SETTINGS 根据应用程序的 config.py 文件中包含的内容，指定微服务中所采用的设置。DATABASE_HOST 变量则设置数据库的访问路由。考察下列代码：

```
famous news service:
     image: famous news service
     build: ./FamousNewsService
     volumes:
```

```
      - '.:/usr/src/app'
    environment:
      - APP SETTINGS=config.DevelopmentConfig
- DATABASE_HOST=mongodb://mongo:27017/
    depends on:
      - mongo
    links:
      - mongo

politics news service:
    image: politics news service
    build: ./PoliticsNewsService
    volumes:
      - '.:/usr/src/app'
    environment:
      - APP SETTINGS=config.DevelopmentConfig
- DATABASE_HOST=mongodb://mongo:27017/
    depends on:
      - mongo
    links:
      - mongo

sports news service:
    image: sports news service
    build: ./SportsNewsService
    volumes:
      - '.:/usr/src/app'
    environment:
      - APP SETTINGS=config.DevelopmentConfig
- DATABASE_HOST=mongodb://mongo:27017/
    depends on:
      - mongo
    links:
      - mongo
```

注意：

上述 3 个微服务指向同一个数据库——它们使用了同一个容器。

此处强烈推荐使用 ongoing 数据的内部控制模式；而命令查询职责分离（CQRS）和缓存优先策略则可以正常方式使用。与物理数据存储组件的数量相比，CQRS 稍显复杂，但缓存机制可与该模式实现较好的匹配。

这里应注意缓存问题。共享数据模式不仅涉及共享数据库的存储，还包括其他物理

存储资源，例如缓存。因此，对于从缓存中删除数据这一类操作，此类行为应得到彻底的处理。

5.4 响应整合

共享数据模式并不能解决所有的应答整合问题，该模式主要集中于存储问题。然而，当对数据库进行访问时，需要对某些 Nginx 配置进行适当调整。

下面将修改 upstream 配置实例。不难发现，此处需要修改 upstream 名称，并向微服务中添加更多服务器实例，如下所示：

```
upstream proxy servers {
      server bookproject userservice 1:3000;
      server bookproject userservice 2:3000;
      server bookproject userservice 3:3000;
      server bookproject userservice 4:3000;
      server bookproject famous news service 1:5000;
      server bookproject famous news service 2:5000;
      server bookproject famous news service 3:5000;
      server bookproject famous news service 4:5000;
      server bookproject politics news service 1:5000;
      server bookproject politics news service 2:5000;
      server bookproject politics news service 3:5000;
      server bookproject politics news service 4:5000;
      server bookproject sports news service 1:5000;
      server bookproject sports news service 2:5000;
      server bookproject sports news service 3:5000;
      server bookproject sports news service 4:5000;
}
```

在更改了 upstream 方法配置后，还需修改 proxy_pass 进而使用 upstream，如下所示：

```
location / {
            proxy_pass http://proxy_servers;
```

5.5 微服务通信

截至目前，全部微服务均可通过 HTTP 协议直接加以访问。利用共享数据模式，无须对其进行调整。目前，所有的微服务通信均声明于 Nginx 配置文件中。

考虑到我们对应用程序采取了渐进式开发过程，因而在后续章节中，还将对微服务通信自身加以讨论。

5.6　存储共享反模式

共享数据模式对于绿色项目来说可视作一种反模式，因而不应视为最终的解决方案，甚至对遗留项目而言也是如此。在第 4 章中，我们曾看到物理组件肩负过多职责所带来的后果。

理想的方法是针对数据共享寻找一种较为优雅的方案，例如改进微服务间的通信；对端点实现专有化，以接收全部业务信息；对于信息共享，系统消息队列仅包含少量选项。

5.7　最 佳 实 践

存储是应用程序不可或缺的一项功能。然而，不正确的使用方式往往会对应用程序带来各种问题，其中也包括微服务。

当考察诸如共享数据这一类模式应用时，应遵循某些最佳实践方案，其中包括：

- 数据库是用来存储数据的，而非业务规则。在数据库中存储业务规则是一种错误的做法，这将使得应用程序依赖于某种结构、缓存实现，并阻碍数据迁移过程以及分布处理。
- 数据库用于数据存储，而非通信事件。一些开发团队采用包含数据库资源的触发器过程，或者利用 worker 查看存储信息的变化。这里的问题是，此类触发器难以监视和调试，同时也是一种获取存储业务规则的方法。
- 不要创建包含循环依赖关系的实体。毫无疑问，这可视为存储中最大的问题。如果不对域进行重构的话，将无法迁移至独立的数据库中。

如果读者忽略了上述问题，当采用其他模式时，应用程序的开发过程将变得异常艰难。

5.8　测 试 机 制

鉴于共享数据模式主要关注存储层，因而我们将拥有极其简单的测试层，其中单元测试和功能测试已然足够。下面考察一个简单的例子。

在 FamousNewsService tests.py 文件中，可运行多项测试。首先需要声明 import 语句。

此处须关注 flask_testing，这也是全部测试过程的基础。该工具提供了一组功能，包括如何访问 Flask 和 HTTP 客户端的设置内容，如下所示：

```
import json
import unittest
from app import app
from flask_testing import TestCase
```

接下来针对功能测试编写基类。针对当前任务，可采用 TestingConfig 设置，如下所示：

```
class BaseTestCase(TestCase):

    def create app(self):
        app.config.from_object('config.TestingConfig')
        return app
```

随后声明配置层的单元测试，如下所示：

```
class TestDevelopmentConfig(TestCase):
    def create app(self):
        app.config.from object('config.DevelopmentConfig')
        return app
    def test app is development(self):
        self.assertTrue(app.config['DEBUG'] is True)
class TestTestingConfig(TestCase):
    def create app(self):
        app.config.from object('config.TestingConfig')
        return app

    def test app is testing(self):
        self.assertTrue(app.config['DEBUG'])
        self.assertTrue(app.config['TESTING'])

class TestProductionConfig(TestCase):
    def create app(self):
        app.config.from object('config.ProductionConfig')
        return app

    def test app is production(self):
        self.assertFalse(app.config['DEBUG'])
        self.assertFalse(app.config['TESTING'])
```

作为示例，下面编写一个路由测试，并于其中创建 News 微服务。由于将通过端点访问应用程序，并检测所接收的结果，因而这是一类功能测试。需要注意的是，执行 POST

的客户端位于当前实例中，通过继承机制，该实例源自 flask_testing，如下所示：

```
class TestNewsService(BaseTestCase):
    def test add news(self):
        """Ensure a new user can be added to the database."""
        with self.client:
            response = self.client.post(
                '/famous/news',
                data=json.dumps(dict(
                    title='My Test',
                    content='Just a service test',
                    author='unittest',
                    tags=['Test', 'Functional test'],
                )),
                content type='application/json',
            )
            data = json.loads(response.data.decode())
            self.assertEqual(response.status code, 201)
            self.assertIn('success', data['status'])
            self.assertIn('My Test', data['news']['title'])
```

最后，还需编写相关代码，并执行来自 Python 调用中的测试，如下所示：

```
if   name   == '  main  ':
    unittest.main()
```

鉴于测试过程未涉及模式问题，因而此处也不打算对此加以深入讨论。在后续章节中，我们还将学习其他模式，应用程序也在此基础上逐步予以完善。

5.9　共享数据模式的利弊

作为一种反模式，共享数据模式受到许多人的喜爱，但其他人认为这是一个不应该再应用的旧概念。然而，理想世界和现实世界间总是存在相当大的距离。

理想状态下，所有的项目都是绿色项目，且无须处理遗留代码或更改单体应用程序的架构。然而，实际情况并非如此。如果遗留项目现在正在为用户服务，这些应用程序包括在线商店、金融应用程序、社交网络和大量需要升级以实现自动化、可扩展性和弹性的业务。

共享数据模式搜索加快了这些遗留应用程序的修改过程，并给开发团队留与一定的时间，以将信息从数据库中分离出来，并评估数据的一致性。就这一方面来说，共享数

据模式具备一定的优势。

另外，共享数据模式还可帮助公司重置系统的架构。

但是，由于全部微服务均置于同一存储中，因而该模式至少蕴含了一种风险。

可以肯定的是，共享数据模式可以看作是一种不能使用的反模式。最后，开发团队负责理解软件的功能和需求内容。

5.10 本章小结

本章创建了 News 微服务并将其与数据库进行连接。对于单体和微服务间的迁移，本章还介绍了一种十分有用的模式。

如果问题围绕着应用程序扩展展开，那么共享数据模式暂且是一种较好的选择方案。此外，其他问题还会涉及存储层，第 6 章将讨论如何解决这个问题。

第 6 章　聚合器微服务设计模式

第 5 章介绍了共享数据模式设计的操作和应用。我们已经了解到，该模式是一种临时模式，也就是说，对于单体至微服务间的过渡转换，此类模式在组件上包含了某些风险。本章将采用一种更加一致的模式，对于微服务而言，该模式体现了较好的开发实践行为。

本章将探讨聚合器设计模式。准确地讲，在这一新模式的基础上，我们将针对 News 微服务使用命令查询职责分离（CQRS）和事件源机制。除此之外，还将对存储分布进行重构。因此，本书涵盖了大量的新概念以及相关实践操作。

本章主要涉及以下内容：

- 数据库分离。
- 微服务重构。
- 微服务通信
- 功能测试。
- 集成测试。

6.1　理解聚合器设计模式

聚合器设计模式包含了简单的概念，但取决于具体场合，其应用性可能会较为复杂。当考察真实场景时，将会发现聚合器设计模式是最具可用性和扩展性的模式。

下面考察当前微服务。之前，我们构建了微服务以操控源自的 UsersService 数据，以及其他 3 个微服务操控来自 News 微服务的数据。

当处理 UsersService 时，可以说，截至目前，该微服务自身已可满足要求。其中，微服务的业务十分简单，并由数据的注册和显示构成。除此之外，再无其他业务需求可促使我们修改该项微服务。

对于 News 微服务来说，情况则有所不同。其中，微服务的客户端之一是门户网站的主界面；针对该客户端，还可显示消息混合结果。为了满足这一条件，我们需要使用聚合器设计模式。

图 6.1 显示使用聚合器设计模式后，应用程序的行为方式。负责 Users 数据的微服务通过负载均衡器/代理直接访问，但负责 News 的微服务则在负载平衡器之前添加了一个新的交互层。该 News 层的工作方式类似于编排器。

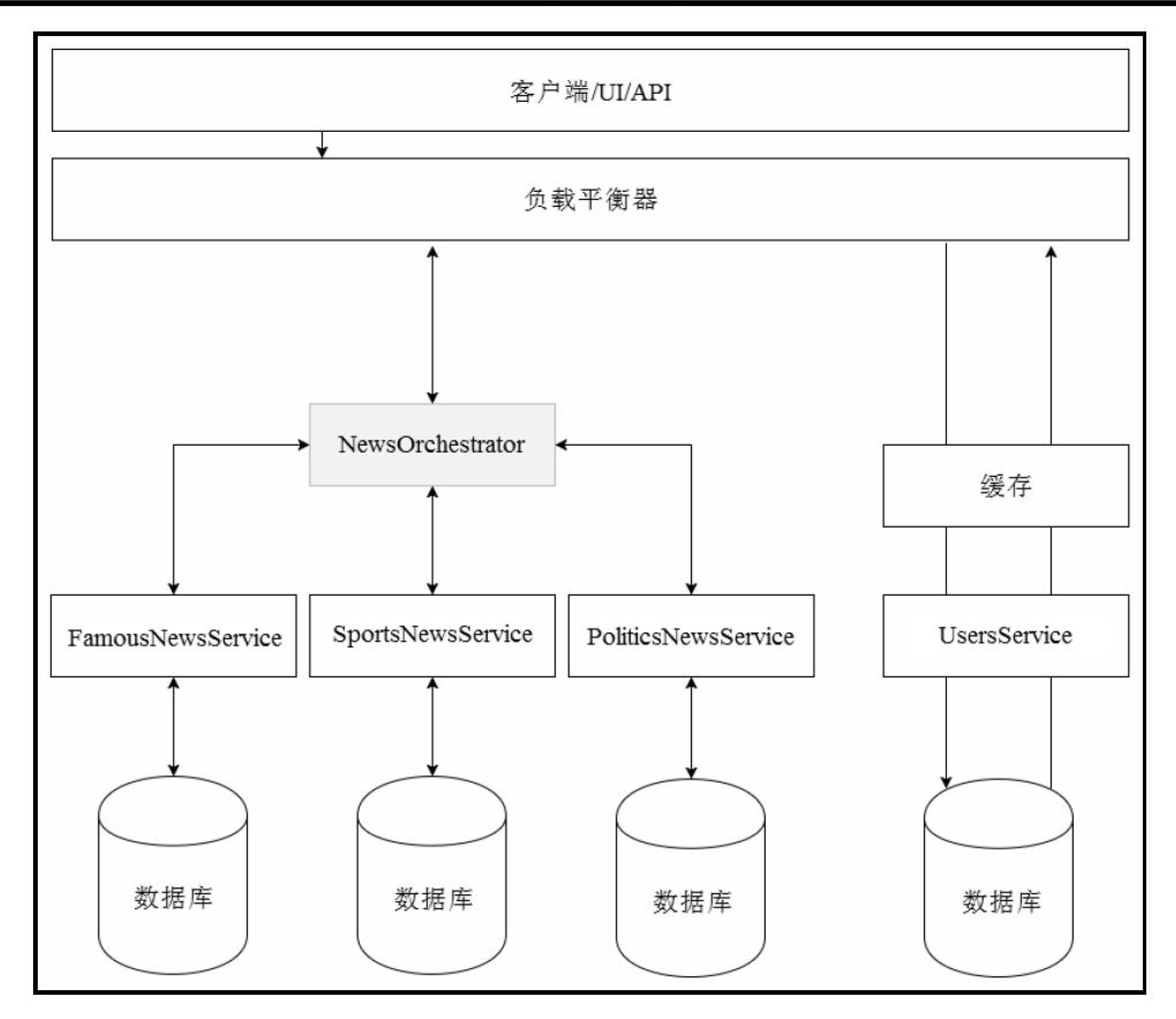


图 6.1

下面考察一些来自客户端的需求内容。在当前客户端中，UI 需要获取来自 3 个不同 News 的信息。为了完成此项任务，UI 须了解每种 News 类型的路由，识别最佳方式以收集接收到的数据，并针对后端至少生成 3 个不同的请求。该过程属于 Web 客户端的职责范畴。下面将简单地创建一个应用程序，但该程序不能满足微服务真实的应用需求。

为了对此类问题进行修正，我们需要采用聚合器设计模式。利用该模式，可创建一个微服务，负责编排客户端数据，并提供该信息的单一访问点。最终，基于 UI 的请求数量将从 3 个减为 1 个。

通过这一方式，还将使用到一个新的概念，其中包含了内部服务（生成和提供数据的微服务）以及公共服务（该微服务负责编排多种数据），以及与客户端层有更直接联系的内部服务。

为了正确地使用聚合器设计模式，需要实现以下各项步骤：

（1）隔离共享数据库。

（2）调整通信层。

（3）创建微服务数据编排器。

随后，即可使用聚合器这一新型的设计模式。注意，我们不止是分离数据并创建一个编排器，还需使用到 CQRS 和事件源。

这两种内部模式与 News 应用程序紧密相关。需要记住的是，一个著名的新闻门户网站需要有良好的读者声誉，而事件来源是审计新闻内容以避免错误和内容撤换的最佳途径。

CQRS 非常适用于此类微服务，其原因在于，我们的核心业务是以同样的方式提供内容，同时始终准备推出最新的内容。这意味着，我们必须有效地记录新数据，并显示所记录的数据。利用 CQRS，可通过分别优化写入层和读取数据来实现这一功能。

6.2　使用 CQRS 和事件源

当前，存在 3 个微服务负责提供应用程序中的新闻内容，即 famous_news_service、sports_news_service 和 politics_news_service。每项微服务均包含了相同的技术结构，但被不同的领域所分隔。由于每项微服务的领域均已设定，因而可采纳完全不同的技术规划。然而，在当前示例中，我们将以相同方式调整 3 项微服务，并使用聚合器设计模式。第一处修改将在存储层中进行。

6.2.1　分离数据库

在每个微服务目录中，我们将生成两个目录，其中均设置了包含数据库配置的 Dockerfile。另外，此处还将针对每个微服务采用两个不同的数据库以使用 CQRS。第一个数据库用于 CommandStack，而第二个数据库则用于 QueryStack。

通过上述调整，News 微服务的目录结构如下所示：



```
├──FamousNewsService
│   ├──command db
│   └──query db
├──PoliticsNewsService
│   ├──command db
│   └──query db
└──SportsNewsService
    ├──command db
    └──query_db
```

对此，我们仅展示了 famous_news_service 微服务中的代码变化，此类修改内容也适用于其他两个 News 微服务。

command_db 目录中包含了两个文件，即 Dockerfile 和 create.sql 文件。

1．编写 CommandStack 容器

command_db 目录中针对 CommandStack 容器定义了 Dockerfile。Dockerfile 包含了环境变量，以及负责生成数据库的初始文件。考察下列文件内容：

```
FROM postgres

ENV POSTGRES_USER=postgres
ENV POSTGRES_PASSWORD=postgres

# run create.sql on init
ADD create.sql /docker-entrypoint-initdb.d
```

2．创建新闻数据库

在 command_db/create.sql 文件中，定义了产品、开发和测试数据库，如下所示：

```
CREATE DATABASE news_prod;
CREATE DATABASE news_dev;
CREATE DATABASE news_test;
```

3．编写 QueryStack 容器

在设置了 CommandStack 文件后，下面在 QueryStack 中创建文件。

在 query_db 目录中，仅包含一个 Dockerfile 文件。在该文件中，将 MongnDB 配置为当前数据库。考察下列内容：

```
FROM mongo:latest
CMD [ "mongod", "--smallfiles", "--logpath=/dev/null" ]
```

在数据库配置设置项尾部，分别设置了基于 CommandStack 的 PostgreSQL，以及基于 QueryStack 的 MongoDB。这种选择结果与业务所涉及的两种技术有关。Postgres 处理的是一致性较为重要的区域，而 Mongo 处理的则是非阻抗（non-impedance）相对重要的区域。

6.2.2　重构微服务

第 1 章中曾谈到，微服务的生命周期较短，当产生业务需求时，即需要对其进行调整——当前即面临着此类问题。

考虑到业务需求，我们将采取新的模式，它将对堆栈和微服务产生直接影响。这里，重要的是理解 News 微服务并不包含外部通信，它们仅是内部服务。另外，不应在内部微服务之间的通信中使用 HTTP。

News 微服务构建于 Flask 框架上，同时包含了与客户端直接对话的 API。当前，这一机制将被完全修改。我们将使用一个新的消息传递系统，即消息代理负责通信。另外，在微服务组成方面的主要框架将使用 Nameko。

1. 选择需求

下面将开始定义微服务依赖关系。在 requirements.txt 文件中，将 Nameko 设置为当前框架，将 SQLAlchemy 设置为 ORM，针对 MongoDB 访问设置 MongoEngine，并设置 Postgres 驱动程序以及 PyTest 以供测试所用。对应文件内容如下所示：

```
nameko
nameko-sqlalchemy
mongoengine
sqlalchemy
psycopg2
pytest
```

2. 配置框架

针对 Nameko 的各项功能，config.yaml 文件定义了相关配置。首先，需要通知 Nameko 针对消息代理的访问路由，在当前示例中为 RabbitMQ，如下所示：

```
AMQP_URI: 'amqp://guest:guest@rabbitmq'
```

接下来，将传递访问数据库的路径，从而支持基于 Nameko 的依赖注入，如下所示：

```
DB_URIS:
   "command_famous:Base": ${COMMANDDB_DEV_HOST}
```

下一个设置是应用程序所使用的日志级别，如下所示：

```
LOGGING:
   version: 1
   handlers:
         console:
            class: logging.StreamHandler
   root:
      level: DEBUG
      handlers: [console]
```

完整的 config.yaml 文件包含了以下配置内容：

```
AMQP URI: 'amqp://guest:guest@rabbitmq'

DB URIS:
   "command famous:Base": ${COMMANDDB DEV HOST}

LOGGING:
   version: 1
   handlers:
      console:
         class: logging.StreamHandler
   root:
      level: DEBUG
      handlers: [console]
```

3．配置容器

当 config.yaml 和 requirements.txt 设置完毕后，下面开始调整 Dockerfile。requirements.txt 文件的执行方式与之前基本一致，唯一的差别在于 ENTRYPOINT 和 CMD——二者将使用 Nameko 框架。除此之外，在 CMD 中，还传递了刚刚创建的 config.yaml 文件，如下所示：

```
FROM python:3.6.1
COPY . /app
WORKDIR /app
RUN pip install -r requirements.txt
ENTRYPOINT ["nameko"]
CMD ["run", "--config", "config.yaml", "service"]
EXPOSE 8000
```

4．编写模型

当实现 CQRS 时，我们针对数据库设置了两种实体表达方式，分别服务于 CommandStack 和 QueryStack。下面将创建 models.py 文件。

与往常的 Python 文件一样，首先是导入依赖项。具体来说，此处需要导入本地库依赖项和 MongoEngine，例如字段类型和用于连接模型和 MongoDB 实例的 connect 函数，如下所示：

```
import os
from datetime import datetime
from mongoengine import (
   connect,
   Document,
   DateTimeField,
```

```
    ListField,
    IntField,
    StringField,
)
```

随后，可向 SQLAlchemy 添加导入，其中包含了字段定义，并指定所用的 SQLAlchemy 数据库类型，如下所示：

```
from sqlalchemy import (
    Column,
    String,
    BigInteger,
    DateTime,
    Index,
)
from sqlalchemy.dialects import postgresql
from sqlalchemy.ext.declarative import declarative_base
```

下面利用 SQLAlchemy 和 Postgres 对实体定义执行 CommandStack 操作。此处将使用到事件源。注意，除了唯一的 ID 之外，它们还包含了一个 version 字段。相应地，该 ID 和 version 字段表示为数据库的组合键。据此，一般不会对某一篇文章进行更新，但会包含同一类新闻的新版本，如下所示：

```
Base = declarative base()

class CommandNewsModel(Base):
      tablename   = 'news'

    id = Column(BigInteger, primary key=True)
    version = Column(BigInteger, primary key=True)
    title = Column(String(length=200))
    content = Column(String)
    author = Column(String(length=50))
    created at = Column(DateTime, default=datetime.utcnow)
    published at = Column(DateTime)
    news type = Column(String, default='famous')
    tags = Column(postgresql.ARRAY(String))

    __table_args__ = Index('index', 'id', 'version')
```

作为 CommandStack 实体，还需对 QueryStack 的信息进行版本控制。但在当前情况下，我们会对数据进行更新，并且会一直保留最新的版本，如下所示：

```
connect('famous', host=os.environ.get('QUERYBD HOST'))

class QueryNewsModel(Document):
    id = IntField(primary key=True)
    version = IntField(required=True)
    title = StringField(required=True, max length=200)
    content = StringField(required=True)
    author = StringField(required=True, max length=50)
    created at = DateTimeField(default=datetime.utcnow)
    published at = DateTimeField()
    news type = StringField(default="famous")
    tags = ListField(StringField(max_length=50))
```

最终，完整的文件内容如下所示：

```
import os
from datetime import datetime
from mongoengine import (
    connect,
    Document,
    DateTimeField,
    ListField,
    IntField,
    StringField,
)

from sqlalchemy import (
    Column,
    String,
    BigInteger,
    DateTime,
    Index,
)
from sqlalchemy.dialects import postgresql
from sqlalchemy.ext.declarative import declarative base


Base = declarative base()

class CommandNewsModel(Base):
      tablename    = 'news'

    id = Column(BigInteger, primary_key=True)
```

```
    version = Column(BigInteger, primary key=True)
    title = Column(String(length=200))
    content = Column(String)
    author = Column(String(length=50))
    created at = Column(DateTime, default=datetime.utcnow)
    published at = Column(DateTime)
    news type = Column(String, default='famous')
    tags = Column(postgresql.ARRAY(String))

      table args   = Index('index', 'id', 'version'),

connect('famous', host=os.environ.get('QUERYBD HOST'))

class QueryNewsModel(Document):
    id = IntField(primary key=True)
    version = IntField(required=True)
    title = StringField(required=True, max length=200)
    content = StringField(required=True)
    author = StringField(required=True, max length=50)
    created at = DateTimeField(default=datetime.utcnow)
    published at = DateTimeField()
    news type = StringField(default="famous")
    tags = ListField(StringField(max_length=50))
```

5. 创建服务

如前所述，News 微服务通信层所在的文件称为 view.py 文件。然而，HTTP 通信层并不存在于应用程序中，最终导致 views.py 文件也不位于其中。相反，可定义一个名为 services.py 的新文件，该文件负责构建基于消息代理的通信。同时，service.py 文件也是声明 CommandStack 和 QueryStack 的地方。

一如既往，下面将首先声明依赖关系。在下列代码中，将声明模型和使用 Nameko 所需的一切内容。

```
import mongoengine

from models import (
    CommandNewsModel,
    Base,
    QueryNewsModel,
)

from sqlalchemy import Sequence
```

```
from nameko.events import EventDispatcher
from nameko.rpc import rpc
from nameko.events import event handler
from nameko_sqlalchemy import DatabaseSession
```

根据已声明的导入内容，可编写相应的命令类。其中，前 3 项内容表示为类属性，并定义了命令名、实例化事件分配器以及依赖注入数据库，如下所示：

```
class Command:
    name = 'command famous'
    dispatch = EventDispatcher()
    db = DatabaseSession(Base)
```

然后，我们有一个使用 rpc 装饰器的方法。该装饰器来自 Nameko 框架，并建立了 RPC 通信模型。由于我们采用了内部事件源模式以及 CQRS，因而须注意某些特殊之处。

add_news 方法将是编排微服务中 RPC 调用的一部分内容，该方法不可或缺，它代表了创建新闻的命令。如果我们考虑一个典型的 CRUD 过程，这意味着写入操作是由相同的命令执行的。究其原因，这是因为我们不再执行更新操作。当客户端请求更新操作时，实际上，这会在所需的数据上创建一个历史事件，而不是在数据库上对现有的数据进行更新，如下所示：

```
@rpc
def add_news(self, data):
```

下面将处理新闻添加事件的 ID 和版本。该控制流验证相关新闻是新内容，抑或是现有新闻文章的新版本。据此，仅在 add_news 上生成事件源，如下所示：

```
try:
    version = 1
    if data.get('version'):
        version = (data.get('version') + 1)
    if data.get('id'):
        id = data.get('id')
    else:
        id = self.db.execute(Sequence('news_id_seq'))
```

在 add_news 事件的识别控制之后，将实例化负责在数据库中注册新闻的实体，如下所示：

```
news = CommandNewsModel(
      id=id,
      version=version,
```

```
    title=data['title'],
    content=data['content'],
    author=data['author'],
    published at=data.get('published at'),
    tags=data['tags'],
)
self.db.add(news)
self.db.commit()
```

利用规范化数据库中所定义的注册过程，将处理事件的分派操作，并在非规范化数据库，即 QueryStack 数据库中进行注册。在类开始处声明的分派实例用于向非规范化 DB 发送当前域中的数据。并行地将事件发送到 QueryStack 后，将通过 RPC 调用生成一个新事件，并通知 add_news 命令所发生的内容，如下所示：

```
data['id'] = news.id
data['version'] = news.version
self.dispatch('replicate db event', data)
return data
```

如果处理过程中出现任何问题，将执行标准数据库中的回滚操作。更为重要的是，我们将了解到为何无法在非规范化数据库中执行相同的过程，如下所示：

```
except Exception as e:
   self.db.rollback()
   return e
```

在创建了 CommandStack 层后，下面进入 QueryStack 的开发过程。首先，我们将使用类声明，以及 Nameko 框架的引用名称，如下所示：

```
class Query:
   name = 'query_famous'
```

因此，这里编写了一个处理程序，并监听 CommandStack 分派的事件，如下所示：

```
@event handler('command famous', 'replicate db event')
def normalize_db(self, data):
```

QueryStack 数据库具有一个较为特殊的特征，该数据库并不是 CommandStack 数据库的完全镜像，而是一个专用数据库，仅包含与新闻文章相关的最新数据。这定义为一个总表，并通过索引机制提供了快速的搜索功能。

为了使其成为专用数据库，首先需要查找与新闻文章相关的数据。如果不存在基于搜索新闻数据生成的事件，那么，将创建一个新的新闻纪录。这就是为什么在我们的 CommandStack 层中发生错误时，我们不回滚到 QueryStack 库的主要原因。关键之处在

于，我们一直对数据执行专有化操作，如果出现任何问题，最终的不一致性问题将不再是一类值得关注的问题，尤其是对于以下业务内容：

```
try:
    news = QueryNewsModel.objects.get(
        id=data['id']
    )
    news.update(
        version=data.get('version', news.version),
        title=data.get('title', news.title),
        content=data.get('content', news.content),
        author=data.get('author', news.author),
        published_at=data.get('published_at', news.published_at),
        tags=data.get('tags', news.tags),
    )
    news.reload()
except mongoengine.DoesNotExist:
    QueryNewsModel(
        id=data['id'],
        version=data['version'],
        title=data.get('title'),
        content=data.get('content'),
        author=data.get('author'),
        tags=data.get('tags'),
    ).save()
except Exception as e:
    return e
```

下一步是针对 QueryStack 数据创建访问点。下面编写一个 get_news 方法，该方法包含了来自 Nameko 的 RPC 装饰器且较为简单。这里，将通过唯一的 ID 作为参考并使用 MongoEngine 搜索新闻，如下所示：

```
@rpc
def get_news(self, id):
    try:
        news = QueryNewsModel.objects.get(id=id)
        return news.to_json()
    except mongoengine.DoesNotExist as e:
        return e
    except Exception as e:
        return e
```

随后，将创建另一个 RPC 请求，即针对数据库中注册的所有新闻的分页搜索，如下

所示：

```
@rpc
def get all news(self, num page, limit):
    try:
        if not num page:
            num page = 1
        offset = (num page - 1) * limit
        news = QueryNewsModel.objects.skip(offset).limit(limit)
        return news.to json()
    except Exception as e:
        return e
```

最终，完整的 service.py 文件包含以下格式：

```
import mongoengine

from models import (
    CommandNewsModel,
    Base,
    QueryNewsModel,
)

from sqlalchemy import Sequence

from nameko.events import EventDispatcher
from nameko.rpc import rpc
from nameko.events import event handler
from nameko sqlalchemy import DatabaseSession

class Command:
    name = 'command famous'
    dispatch = EventDispatcher()
    db = DatabaseSession(Base)

    @rpc
    def add news(self, data):
        try:
            version = 1
            if data.get('version'):
                version = (data.get('version') + 1)
            if data.get('id'):
                id = data.get('id')
```

```
            else:
                id = self.db.execute(Sequence('news id seq'))

            news = CommandNewsModel(
                id=id,
                version=version,
                title=data['title'],
                content=data['content'],
                author=data['author'],
                published at=data.get('published at'),
                tags=data['tags'],
            )
            self.db.add(news)
            self.db.commit()
            data['id'] = news.id
            data['version'] = news.version
            self.dispatch('replicate db event', data)
            return data
        except Exception as e:
            self.db.rollback()
            return e

class Query:
    name = 'query famous'

    @event handler('command famous', 'replicate db event')
    def normalize db(self, data):
        try:
            news = QueryNewsModel.objects.get(
                id=data['id']
            )
            news.update(
                version=data.get('version', news.version),
                title=data.get('title', news.title),
                content=data.get('content', news.content),
                author=data.get('author', news.author),
                published at=data.get('published at', news.published at),
                tags=data.get('tags', news.tags),
            )
            news.reload()
        except mongoengine.DoesNotExist:
            QueryNewsModel(
```

```
                id=data['id'],
                version=data['version'],
                title=data.get('title'),
                content=data.get('content'),
                author=data.get('author'),
                tags=data.get('tags'),
            ).save()
        except Exception as e:
            return e

    @rpc
    def get_news(self, id):
        try:
            news = QueryNewsModel.objects.get(id=id)
            return news.to_json()
        except mongoengine.DoesNotExist as e:
            return e
        except Exception as e:
            return e

    @rpc
    def get_all_news(self, num_page, limit):
        try:
            if not num_page:
                num_page = 1
            offset = (num_page - 1) * limit
            news = QueryNewsModel.objects.skip(offset).limit(limit)
            return news.to_json()
        except Exception as e:
            return e
```

在结束了 service.py 文件后，我们将持有 4 个功能型微服务。famous_news_service 中的变化内容也可复制至其他 News 微服务中。下一步是创建数据编排器。

6. 筹备数据库容器以协同工作

为了使当前项目作为 Docker 容器运行，并使用满足 CQRS 的新型数据库，需要对项目的 docker-compose.yml 进行某些调整。再次强调，此处仅针对 famous_news_service 进行修改。然而，相关变化应适应于所有的 News 微服务。

首先，针对当前数据库创建容器。我们已经了解到，新闻容器中的每个数据库均包含一个 Dockerfile，这也是我们所要用到的内容。具体而言，第一个将要创建的容器将服务于应用程序的 QueryStack。需要注意的是，这里将构建指向 famous_news_service 微服

务的内部 Dockerfile，如下所示：

```
querydb famous:
  image: querydb famous
  build: ./FamousNewsService/query db/
  ports:
    - "5433:5432"
  restart: always
```

第二个创建的容器则针对 CommandStack 进行。类似于 QueryStack 数据库容器，此处将针对内部微服务目录予以构建，如下所示：

```
commanddb famous:
  image: commanddb famous
  build: ./FamousNewsService/command db/
  ports:
    - "27017:27017"
  restart: always
  healthcheck:
    test: exit 0
```

下面将重新配置微服务并使用数据库中的容器。其间，构建过程没有发生任何变化。相应地，变化来自环境变量（执行刚刚创建的容器中的数据库）。另一个需要注意的地方是应用于微服务上的依赖关系，如下所示：

```
famous news service:
  image: famous news service
  build: ./FamousNewsService
  volumes:
    - './FamousNewsService:/app'
  environment:
    - QUERYBD HOST=mongodb://querydb famous:27017/
    - QUEUE HOST=amqp://guest:guest@rabbitmq
    - COMMANDDB HOST=postgresql://postgres:postgres@commanddb famous:
5432/news prod?sslmode=disable
    - COMMANDDB DEV HOST=postgresql://postgres:postgres@commanddb famous:
5432/news dev?sslmode=disable
    - COMMANDDB TEST HOST=postgresql://postgres:postgres@commanddb famous:
5432/news test?sslmode=disable
  depends on:
    - querydb famous
    - commanddb famous
    - rabbitmq
```

```
links:
  - querydb famous
  - commanddb famous
  - rabbitmq
```

6.3　微服务通信

本书的主题是讨论微服务间的通信，对此，较好的方法是通过实操方式予以展示。当开发 orchestrator_news_service 微服务时，我们将进一步理解这种通信机制。

本节首先介绍通信流的工作方式。在图 6.2 中，UI 生成请求，历经负载平衡器并到达新闻编排器，这也是负责编排新闻数据的微服务。之后，编排器在消息代理（此处为 RabbitMQ）中编写消息，该消息表示 UI 请求编排的数据类型。

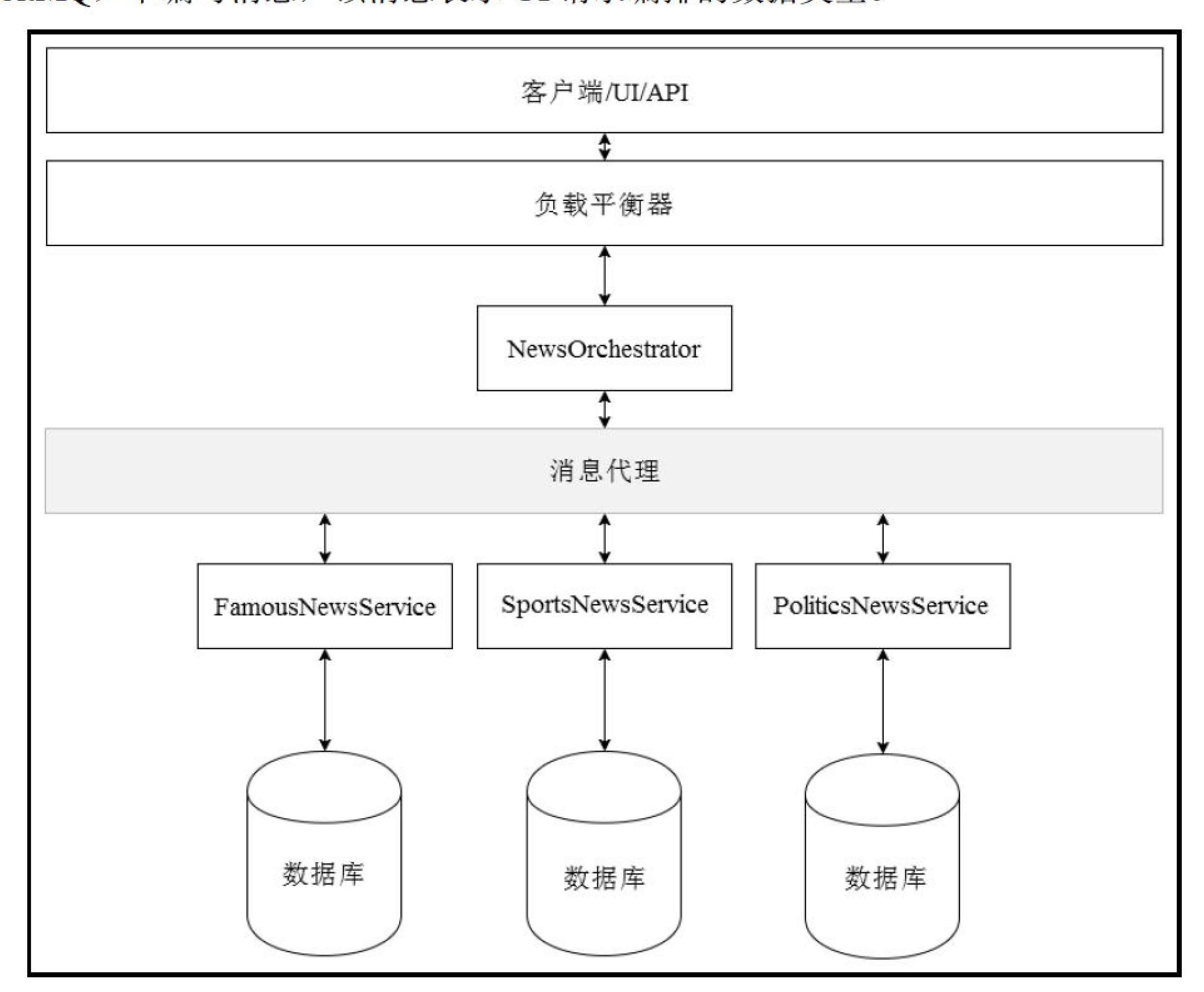


图 6.2

每项 News 微服务均对消息类型有所了解，是否对此予以响应则取决于用户。微服务的响应处理过程类似于编排器，也就是说，微服务也向知晓读取位置的编排器编写响应消息。

这一信息交换过程体现了 New 微服务的通信行为。有些时候，通信通过 RPC 完成；而其他时候，通信过程将生成完全异步和非阻塞事件。

famous_news_service 微服务中即使用了此类通信行为。当在微服务内部使用 CQRS 时，将与这一通信类型协同工作。当在 CommandStack 数据库中发布一些新的内容时，其中包含了两个 RPC 可直接与其他微服务和事件进行会话。当前，唯一的差别在于，我们将把此类通信模型引入系统层面。

6.3.1　创建编排器

orcherstrator_news_servic 的格式与之前讨论的 famous_news_service 类似。这里，编排器表示为一个应用程序，并使用 Flask 作为框架，但不包含任何数据库通信层。该微服务编排器使用的数据不是自身的数据库，而是源自消息代理使用的其他微服务。

1．筹备微服务容器

下面首先创建 Dockerfile，声明编排器容器，并采用与前述微服务相同的扩展标准，如下所示：

```
FROM python:3.6.1

COPY . /app

WORKDIR /app

RUN pip install -r requirements.txt

ENTRYPOINT ["python"]

CMD ["app.py"]

EXPOSE 5000
```

2．编写依赖关系

requirements.txt 文件中的依赖关系十分简单，皆因未使用自身的数据库，如下所示：

```
Flask
Flask-Testing
nameko
```

3. 编写配置文件

类似于之前基于 Flask 的微服务，当前微服务也包含了 config.py 设置文件。需要注意的是，此处并未涉及数据库的访问配置。考察下列代码示例：

```
class BaseConfig:
    """Base configuration"""
    DEBUG = False
    TESTING = False

class DevelopmentConfig(BaseConfig):
    """Development configuration"""
    DEBUG = True

class TestingConfig(BaseConfig):
    """Testing configuration"""
    DEBUG = True
    TESTING = True

class ProductionConfig(BaseConfig):
    """Production configuration"""
    DEBUG = False
```

4. 编写服务器访问

app.py 文件负责创建 Flask 实例，类似于其他 Python 文件，首先需要导入依赖关系。稍后，还将纳入 Flask 实例声明、访问设置、路由声明以及运行服务器的相关命令，如下所示：

```
import os
from flask import Flask

from views import news

# instantiate the app
app = Flask(__name__)

# set config
app_settings = os.getenv('APP_SETTINGS')
```

```
app.config.from_object(app_settings)

# register blueprints
app.register_blueprint(news)

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000)
```

5．创建编排控制器

views.py 依赖关系文件的存在主要是因为导入了 ClusterRpcProxy。对应的依赖关系源自 Nameko 框架，并可提供与其他微服务的连接，如下所示：

```
import os
import json
import itertools
from flask import Blueprint, jsonify, request
from nameko.standalone.rpc import ClusterRpcProxy
```

在导入后，我们将编写 Blueprint 实例，进而确定路由以及基于高级消息队列协议（AMPQ）的消息代理访问，如下所示：

```
news = Blueprint('news', __name__)
CONFIG_RPC = {'AMQP_URI': os.environ.get('QUEUE_HOST')}
```

其中，第一个路由则是通过 ID 的数据搜索。需要注意的是，应向该路由中传递两个参数——第一个参数是新闻类型，第二个参数则是当前 ID。在 get_single_news 中，存在一个函数，可用于处理其他微服务调用，如下所示：

```
@news.route('/<string:news_type>/<int:news_id>', methods=['GET'])
def get_single_news(news_type, news_id):
    """Get single user details"""
    try:
        response_object = rpc_get_news(news_type, news_id)
        return jsonify(response_object), 200
    except Exception as e:
        error_response(e, 500)
```

第二个路由则是 news_service_orchestrator 存在的实际原因。当微服务的使用者需要同时获得来自全部路由的信息时，该路由将发挥功效。另外，该路由表示为全部新闻上的页面搜索结果。重点是为每个微服务执行一个 RPC 调用，然后组织数据以在一个响应中对其予以返回，如下所示：

```
@news.route(
    '/all/<int:num page>/<int:limit>',
    methods=['GET'])
def get all news(num page, limit):
    try:
        response famous = rpc get all news(
            'famous',
            num page,
            limit
        )
        response politics = rpc get all news(
            'politics',
            num page,
            limit
        )
        response sports = rpc get all news(
            'sports',
            num page,
            limit
        )
    # Summarizing the microservices responses in just one
        all news = itertools.chain(
            response famous.get('news', []),
            response politics.get('news', []),
            response sports.get('news', []),
        )
        response object = {
            'status': 'success',
            'news': list(all news),
        }
        return jsonify(response object), 200
    except Exception as e:
        return erro_response(e, 500)
```

第三个路由也是页面搜索，但面向的是每种 News 类型，如下所示：

```
@news.route(
    '/<string:news type>/<int:num page>/<int:limit>',
    methods=['GET'])
def get all news per type(news type, num page, limit):
    """Get all users"""
    try:
        response_object = rpc_get_all_news(
```

```
            news_type,
            num_page,
            limit
        )
        return jsonify(response_object), 200
    except Exception as e:
        return erro_response(e, 500)
```

第 4 个路由将接收 POST 或 PUT，并发送对应微服务的最新新闻文章。其中，对有效载荷过程的解释类似于 Flask，如下所示：

```
@news.route('/<string:news_type>', methods=['POST', 'PUT'])
def add_news(news_type):
    post_data = request.get_json()
    if not post_data:
        return erro_response('Invalid payload', 400)
    try:
        response_object = rpc_command(news_type, post_data)
        return jsonify(response_object), 201
    except Exception as e:
        return erro_response(e, 500)
```

当前，可通过一些辅助函数“查看”工作结果。此处，第一个函数是 error_response，该函数优化重复代码，并以一种较为友好的方式返回错误消息，如下所示：

```
def error_response(e, code):
    response_object = {
        'status': 'fail',
        'message': str(e),
    }
    return jsonify(response_object), code
```

其他 3 个辅助函数则用于确定哪一个函数用于执行 RPC 调用。相应地，rpc_get_news、rpc_get_all_news 和 rpc_command 均包含了相似的逻辑，并可用于调用 ClusterRpcProxy，进而构建与深层次服务间的连接，如下所示：

```
def rpc_get_news(news_type, news_id):
    with ClusterRpcProxy(CONFIG_RPC) as rpc:
        if news_type == 'famous':
            news = rpc.query_famous.get_news(news_id)
        elif news_type == 'sports':
            news = rpc.query_sports.get_news(news_id)
        elif news_type == 'politics':
```

```
            news = rpc.query politics.get news(news id)
        else:
            return erro response('Invalid News type', 400)
        return {
            'status': 'success',
            'news': json.loads(news)
        }

def rpc get all news(news type, num page, limit):
    with ClusterRpcProxy(CONFIG RPC) as rpc:
        if news type == 'famous':
            news = rpc.query famous.get all news(num page, limit)
        elif news type == 'sports':
            news = rpc.query sports.get all news(num page, limit)
        elif news type == 'politics':
            news = rpc.query politics.get all news(num page, limit)
        else:
            return erro response('Invalid News type', 400)
        return {
            'status': 'success',
            'news': json.loads(news)
        }

def rpc command(news type, data):
    with ClusterRpcProxy(CONFIG RPC) as rpc:
        if news type == 'famous':
            news = rpc.command famous.add news(data)
        elif news type == 'sports':
            news = rpc.command sports.add news(data)
        elif news type == 'politics':
            news = rpc.command politics.add news(data)
        else:
            return erro response('Invalid News type', 400)
        return {
            'status': 'success',
            'news': news,
        }
```

6.3.2　使用消息代理

orchestrator_news_service 微服务向内部服务（位于消息代理之后）发送消息。随后

将返回运行于公共服务中的消息并代表内部服务，此处为 orchestrator_news_service。但实际上，该过程未产生任何操作，其原因在于：当前尚未筹备微服务容器。下面将编辑 docker-compose.yml 文件，并包含必要的实例。

1．与容器协同工作

首先需要删除 docker-compose.yml 文件中原有的、在共享设计模式中共有的 MongoDB 实例。由于每个 News 微服务均包含自身的数据库，因而该实例并无太多用处。下列代码将移除原有代码：

```
mongo:
image: mongo:latest
container name: "mongodb"
ports:
- 27017:27017
command: mongod --smallfiles --logpath=/dev/null # --quiet
```

在移除了 Mongo 容器后，下面将创建 RabbitMQ 容器。对于当前微服务来说，这将作为消息代理。

在项目的根目录中，将创建名为 queue 的新目录。在 queue 目录内，可创建包含以下配置信息的 Dockerfile。

```
FROM rabbitmq:3-management
ENV RABBITMQ ERLANG COOKIE: "random string"
ENV RABBITMQ DEFAULT USER: "guest"
ENV RABBITMQ DEFAULT PASS: "guest"
ENV RABBITMQ_DEFAULT_VHOST: "/"
```

待 Dockerfile 创建完毕后，将再次返回至 docker-compose.yml 文件中，并编写配置容器。容器 RabbitMQ 显示了两部分内容——通信工具所使用的 5672 端口，以及用于访问 RabbitMQ 中管理工具的 15672 端口，如下所示：

```
rabbitmq:
   image: rabbitmq
   build: ./queue
   ports:
      - "15672:15672"
      - "5672:5672"
   restart: Always
```

下面将设置编排器实例，该配置的重点内容在于基于 News 微服务的依赖关系声明以及消息代理，如下所示：

```
orchestrator news service:
    image: orchestrator news service
    build: ./NewsOrchestrator
    volumes:
      - './NewsOrchestrator:/app'
    environment:
      - APP SETTINGS=config.DevelopmentConfig
      - QUEUE HOST=amqp://guest:guest@rabbitmq
    depends on:
      - famous news service
      - politics news service
      - sports news service
      - rabbitmq
    links:
      - famous news service
      - politics news service
      - sports news service
      - rabbitmq
```

2．更新代理/复杂平衡器

现在是对应用程序进行重大更改的时候了。当谈及重大变化时，此处并未涉及太多代码级别方面的内容，而是指对应用程序业务以及系统健康的影响程度。

首先需要移除之前生成的上游内容。其中，服务器的配置尚存在一些问题，例如路由冲突以及错误的路由机制，如下所示：

```
upstream proxy servers {
        server bookproject userservice 1:3000;
        server bookproject userservice 2:3000;
        server bookproject userservice 3:3000;
        server bookproject userservice 4:3000;
        server bookproject famous news service 1:5000;
        server bookproject famous news service 2:5000;
        server bookproject famous news service 3:5000;
        server bookproject famous news service 4:5000;
        server bookproject politics news service 1:5000;
        server bookproject politics news service 2:5000;
        server bookproject politics news service 3:5000;
        server bookproject politics news service 4:5000;
        server bookproject sports news service 1:5000;
        server bookproject sports news service 2:5000;
        server bookproject_sports_news_service_3:5000;
```

```
        server bookproject sports news service 4:5000;
}
```

在移除了原有的上游内容时，将针对 users_service 和 orchestrator_news_service 构建最新内容。通过这种方式，将创建完全独立的路由。需要注意的是，在记录每项微服务上游内容之前，由于编排器微服务可采用智能方式处理数据，因而这种工作方式不再必要。这里，仅编排器显示于微服务的使用者，如下所示：

```
upstream users servers {
    server bookproject usersservice 1:3000;
    server bookproject usersservice 2:3000;
    server bookproject usersservice 3:3000;
    server bookproject usersservice 4:3000;
}

upstream orchestrator servers {
    server bookproject orchestrator news service 1:5000;
    server bookproject orchestrator news service 2:5000;
    server bookproject orchestrator news service 3:5000;
    server bookproject orchestrator news service 4:5000;
}
```

由于包含了不同的上游内容，因而我们将创建不同的位置，进而包含更大的配置灵活性，并解决了路由冲突问题。现在，我们有了一个将请求重定向到 users_servers 上游的位置，以及一个将请求重定向到 orchestrator_servers 上游的位置，如下所示：

```
location / {
    proxy pass          http://users servers/;
    proxy redirect      off;
    proxy set header    Host $host;
    proxy set header    X-Real-IP $remote addr;
    proxy set header    X-Forwarded-For $proxy add x forwarded for;
    proxy set header    X-Forwarded-Host $server name;
}

location /news/ {
    proxy pass          http://orchestrator servers/;
    proxy redirect      off;
    proxy set header    Host $host;
    proxy set header    X-Real-IP $remote addr;
    proxy set header    X-Forwarded-For $proxy add x forwarded for;
    proxy set header    X-Forwarded-Host $server name;
}
```

完整的 nginx.conf 文件内容如下所示：

```
worker processes 4;

events { worker connections 1024; }

http {
   sendfile on;

   upstream users servers {
      server bookproject usersservice 1:3000;
   }

   upstream orchestrator servers {
      server bookproject orchestrator news service 1:5000;
   }

   server {
      listen 80;
      location / {
         proxy_pass          http://users_servers/;
         proxy redirect      off;
         proxy set header    Host $host;
         proxy set header    X-Real-IP $remote addr;
         proxy set header    X-Forwarded-For $proxy add x forwarded for;
         proxy set header    X-Forwarded-Host $server name;
      }

      location /news/ {
         proxy pass          http://orchestrator servers/;
         proxy redirect      off;
         proxy set header    Host $host;
         proxy set header    X-Real-IP $remote addr;
         proxy set header    X-Forwarded-For $proxy add x forwarded for;
         proxy set header    X-Forwarded-Host $server name;
      }
   }
}
```

6.4　模式扩展

聚合器设计模式为应用程序提供了较大的可扩展性，主要是因为每个组件都可以单

独实现扩展。当谈论应用程序组件的可扩展性时，通常意味着可针对每项微服务独立地创建多个不同的实例。图 6.3 真实反映了相应的扩展能力，聚合器设计模式仅支持应用程序某一部分的扩展。

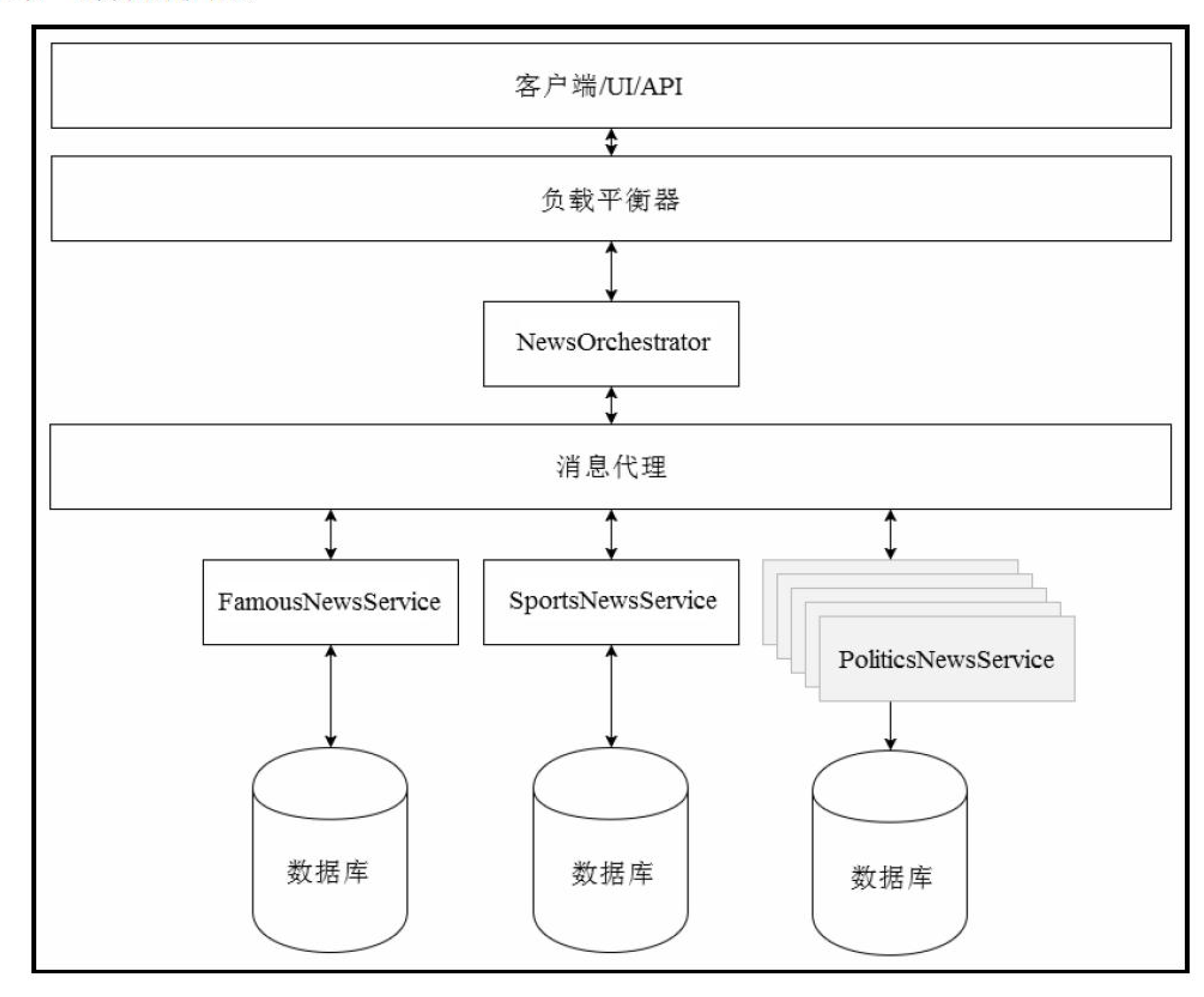


图 6.3

该模式另一个值得注意的地方是分别支持 *x* 轴和 *z* 轴扩展。回忆一下，编排器也是一项微服务，因而可通过缓存机制和其他资源进一步隔离内部服务。

6.5　瓶颈反模式

如前所述，对于可扩展性而言，聚合器设计模式十分高效。但稍有疏忽，该模式将会提供一种反模式。这里，所生成的反模式称作瓶颈。下面考察此类反模式的创建方式。

News 微服务中的应用程序被划分为面向公共的服务以及内部服务。根据这一设计方

式，当访问内部服务时，须经历面向公共的服务，但问题也就此产生。

当工程师们错误地理解应用程序的压力点以及扩展位置时，即会产生瓶颈。考察下列场景：人们需要了解世界棒球大赛的最新新闻。显然，此时 sports_news_service 将接收大量的负载访问。一种较为自然的处理方式是创建更多的 sports_news_service 实例。然而，即使针对 sport_news_service 采用更多的资源，应用程序仍无法扩展。图 6.4 显示了该应用程序当前所处的情况。

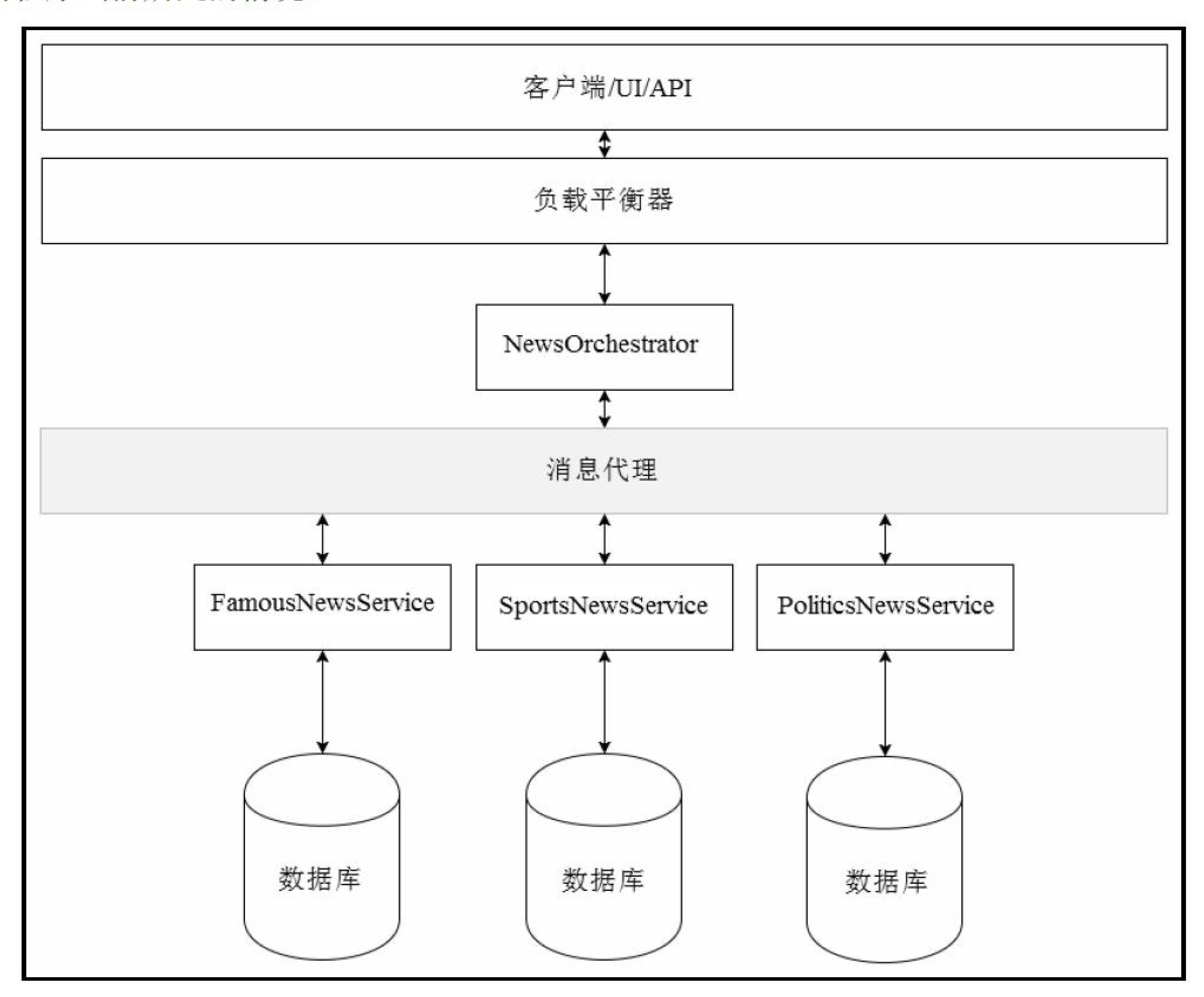


图 6.4

问题来源于我们增加了内部服务的资源，而非面向公共的服务。这意味着，sport_news_service 无法实现正确的响应，其原因在于，微服务路径被阻塞或者性能较差。某些时候，该问题可能会表现得更加严重——面向公共的服务提供了多个微服务，应用程序的其他部分也面临着同样的缓慢问题。利用聚合器设计模式所造成的这种局面可视作是一种瓶颈。

为了解决这一问题，应适当地对应用程序进行扩展，如图 6.5 所示。

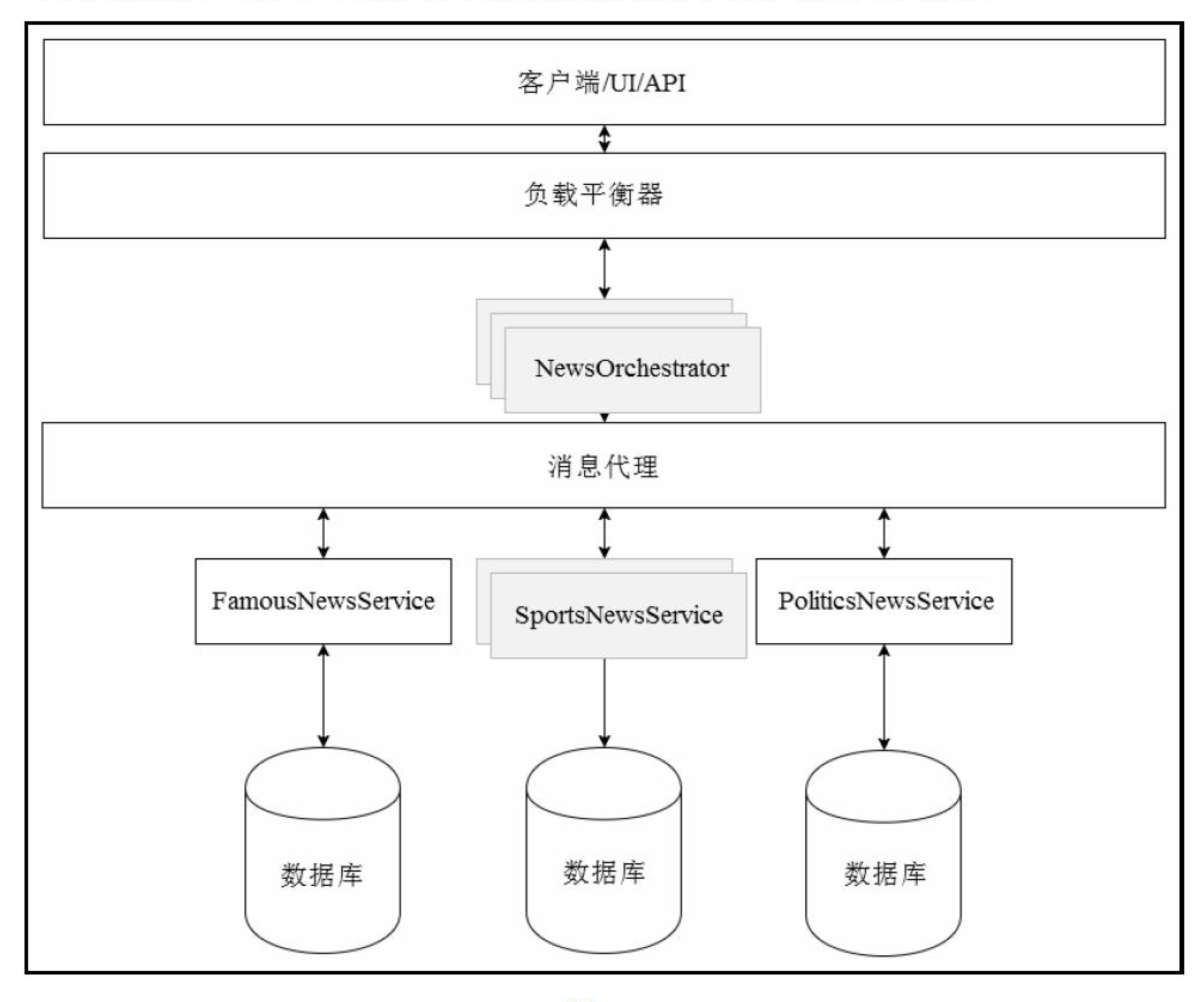


图 6.5

6.6 最 佳 实 践

在本章中，我们已经尝试使用了下列与微服务相关的最佳实践方案：

- 分离数据库可较好地对应用程序进行扩展，尤其是数据存储层。
- 封装微服务可将微服务划分为两个层，即面向公共服务和内部服务。这一类划分方案对于签名微服务来说提供了更大的灵活性。此时，内部服务可更加方便地进行调整。
- 使用 CQRS。当采用 CQRS 时，将移除应用程序中一些不必要的压力点。
- 使用事件源。当使用事件源时，可从新闻文章中处理信息流，这也使我们对每

一篇新闻文章的历史有了一个真实的认识。

- 使用可扩展的模式。利用强大的设计模式，并理解聚合模式的扩展方式，我们将对如何避免反模式有着一个清晰的认知。

前面引用的最佳实践只是一些基本的改进方案，并以一种流畅和动态的方式应用于设计中。一些简单的、结构良好的模式应用可较好地反映出这些实践结果。

6.7　测　　试

在当前阶段，测试处理过程接收新的元素集成。我们需要验证是否满足业务运行的最小功能集。对此，存在两种基本的方案，即每项微服务的功能测试，以及微服务的集成测试。

6.7.1　功能测试

功能测试将证明微服务是否可完美地执行其功能项。在此，我们将使用 famous_news_service 微服务作为示例，并于其中编写命令层测试。

首先将在 tests.py 文件中声明导入语句。Nameko 对测试机制提供了较好的支持，并定义了 worker_factory 函数。该函数通过 Nameko 验证相关元素，且无须启用实际的服务器。对应代码如下所示：

```
import os
import pytest
from .service import Command
from nameko.testing.services import worker_factory
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker
```

在声明了导入语句后，将创建一个固定装置，以连接测试数据库。对此，可采用 PyTest。根据这一固定装置，Nameko 将向期望的元素提供该连接，如下所示：

```
@pytest.fixture
def session():
    db_engine = create_engine(os.environ.get('COMMANDDB_TEST_HOST'))
    Session = sessionmaker(db_engine)
    return Session()
```

在定义了连接后，下面将编写测试。注意，在该测试中，将运行命令层两次。我们

这样做是为了验证 CQRS 中的部分内容，就好像事件源流正在工作一样，如下所示：

```
def test command(session):
    data = {
        "title": "title test",
        "author": "author test",
        "content": "content test",
        "tags": [
            "test tag1",
            "test tag2",
        ],
    }
    command = worker factory(Command, db=session)
    result = command.add news(data)
    assert result['title'] == "title test"
    assert result['version'] == 1

    data['id'] = result['id']
    data['version'] = result['version']
    command = worker factory(Command, db=session)
    result = command.add news(data)

    assert result['version'] == 2
```

这一简单的测试验证了当前业务微服务大约 50%的内容。

6.7.2 集成测试

相信大家都很清楚，orchestrator_news_service 微服务是一个空组件，仅用于编排数据。因此，执行单个测试查找故障点并无太大优势。

此处所执行的测试，与微服务使用者的实际行为相关。利用之前在 Flask 中编写的微服务，我们已经创建了相应的测试。

下面将声明测试类，在该类中将创建两个测试。第一个测试将使用 test_add_news，并查看是否可添加始于编排器的新闻文章。第二个测试将使用 get_all_news，并检测是否可获取一篇始于编排器的新闻文章，如下所示：

```
class TestNewsService(BaseTestCase):

    def test add news(self):
        """Test to insert a News to the database."""
        with self.client:
```

```
            response = self.client.post(
                '/famous',
                data=json.dumps(dict(
                    title='My Test',
                    content='Just a service test',
                    author='unittest',
                    tags=['Test', 'Functional test'],
                )),
                content type='application/json',
            )
            data = json.loads(response.data.decode())
            self.assertEqual(response.status code, 201)
            self.assertIn('success', data['status'])
            self.assertIn('My Test', data['news']['title'])

    def test get all news(self):
        """Test to get all News paginated from the database."""
        with self.client:
            test cases = [
                {'page': 1, 'num per page': 10, 'loop couter': 0},
                {'page': 2, 'num per page': 10, 'loop couter': 10},
                {'page': 1, 'num per page': 20, 'loop couter': 0},
            ]
            for tc in test cases:
                response = self.client.get(
                    '/famous/{}/{}'.format(
                        tc['page'], tc['num per page'])
                )
                data = json.loads(response.data.decode())
                self.assertEqual(response.status code, 200)
                self.assertIn('success', data['status'])
                self.assertEqual(len(data['news']) > 0)
                for d in data['news']:
                    self.assertEqual(
                        d['title'],
                        'Title test-{}'.format(tc['loop couter'])
                    )
                    self.assertEqual(
                        d['content'],
                        'Content test-{}'.format(tc['loop couter'])
                    )
                    self.assertEqual(
```

```
                d['author'],
                'Author test-{}'.format(tc['loop couter'])
            )
            tc['loop_couter'] += 1
```

除此之外，还存在一些其他可用的测试方案，后续章节将对此加以深入讨论。

6.8　聚合器设计模式的优缺点

总而言之，聚合器设计模式利大于弊，它是一种非常优雅的可扩展模式，可以应用于几乎所有的微服务场景中。

6.8.1　聚合器设计模式的优点

聚合器设计模式的优点主要体现在：

- x 轴和 y 轴的可扩展性。
- 隧道效应微服务。
- 微服务签名对内部服务的灵活性。
- 对于微服务提供了单一访问点。

6.8.2　聚合器设计模式的缺点

相比较而言，聚合器设计模式的缺点如下：

- 数据编排的复杂性。
- 瓶颈反模式。
- 微服务通信间的延迟。

6.9　本 章 小 结

本章涵盖了一些较为重要的内容。其中，我们对前述方案中的错误进行了修正，并创建了新型微服务以及处于分离状态的容器，同时还展示了多种可扩展应用。除此之外，本章还介绍了聚合器设计模式。

第 7 章将继续探讨微服务设计模式，即代理设计模式。

第 7 章　代理微服务设计模式

第 6 章讨论了聚合器设计模式的功能和应用，这也是微服务中最为常用的模式。本章将继续介绍一种应用广泛的模式，甚至一些软件过程师也对此少有了解，即代理设计模式。

与第 6 章相比，本章涵盖了大量的概念性内容，但极具指导意义。下面探讨代理设计模式的工作方式，以及应用时机。在该处理过程中，我们将查看一些最佳实现方案，以及该模式的一些负面效应。

本章主要涉及以下内容：

- 代理策略。
- 微服务通信。
- 模式的扩展性。

7.1　代 理 方 案

代理设计模式可视为聚合器设计模式的变化版本。当需要组合或封装微服务，或者不需要聚合值时，即可使用代理设计模式。基本上讲，该模式可直接访问业务内容，但会隔离技术层。类似于聚合器设计模式，代理设计模式支持独立扩展，其中包括 x 轴和 y 轴。

由于代理设计模式支持微服务的单点访问，因而可视为一种更纯粹的模式，通常不需要在微服务之间进行通信。另一个与代理设计模式相关的显著特征则是使用代理时与其他模式间的应用灵活性。

通常情况下，作为一种技术决策，代理设计模式并不会对业务层产生影响。唯一的例外是应用程序的使用者可以方便地从一个引用（代理）中获取数据。

在图 7.1 中可以看到，代理负责重定向请求。其中，某个请求针对既定路径而完成，对于特定路径上的搜索信息，应用程序的使用者并不知道微服务的位置。代理负责理解当前请求，并将其传递至微服务中，同时知晓正确地返回期望信息，或者执行某项任务。

基本上讲，代理设计模式包含了两种方案模型，即哑代理（dumb proxy）和智能代理。这两种模型均十分有效且较为常见。但我们需要确定哪一种方案更适用于当前业务。

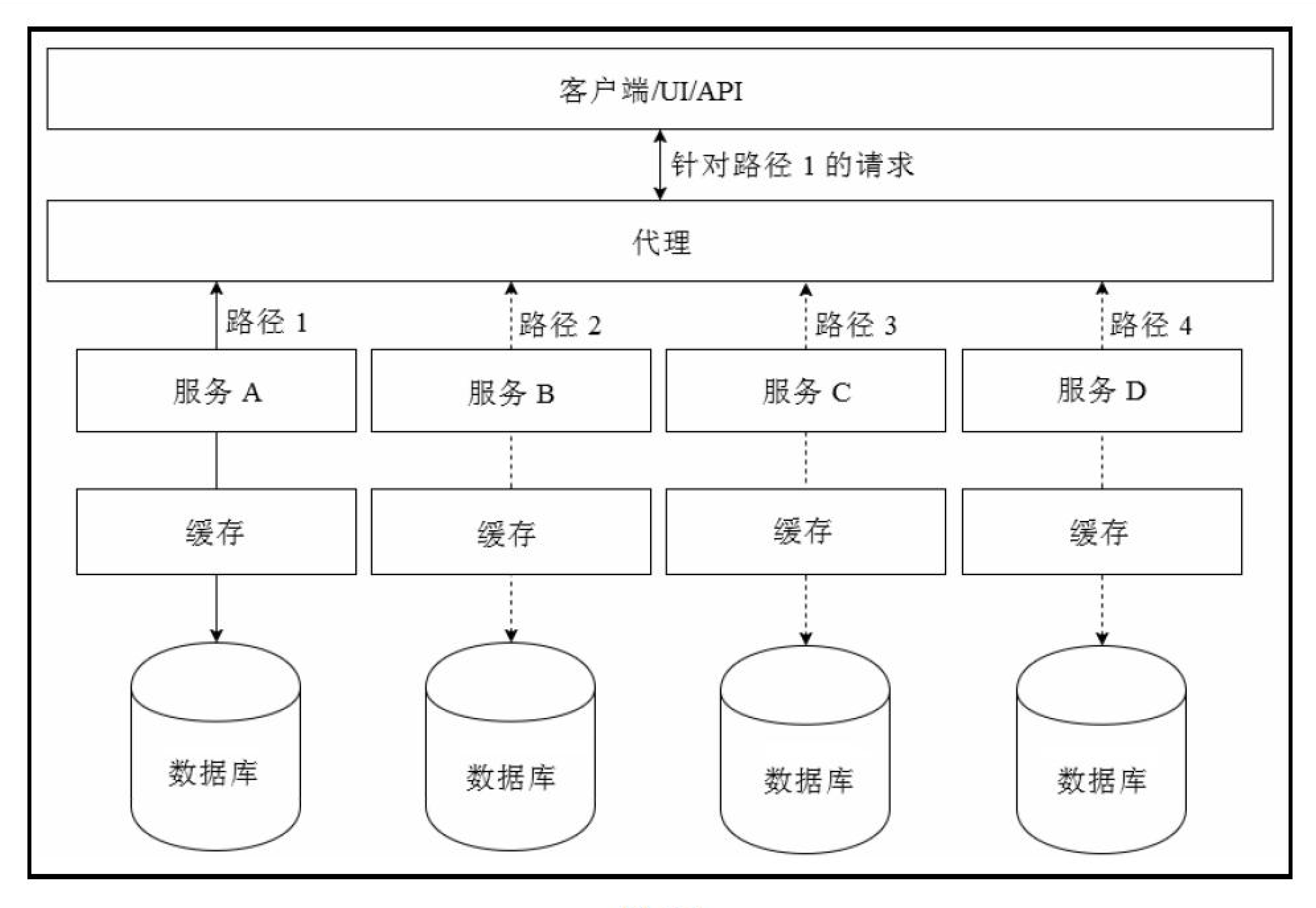


图 7.1

7.1.1　哑代理

顾名思义，哑代理是一种不包含任何智能的模型，其唯一目标是提供单一的端点，以简化应用程序客户端，并封装微服务路径的直接访问操作。

因此，在该策略中，可将请求委托至正确的位置，并提供代理上的、微服务的可扩展性以及可用性。

7.1.2　智能代理

智能代理这一称谓源自可执行更多项任务，而不是将请求委托至其微服务中。使用智能代理策略可以执行一系列简单的任务，而内容修改则是智能代理执行的最常见的任务。

想象一下，微服务响应了某个应用程序客户端，该过程是通过微服务响应中的额外字段完成的，但不会进入另一个应用程序客户端。许多人会说，理想的方法是创建同一个 API 的两个版本，以满足两个不同的消费者。一种常见的方法是修改代理级响应。通过这种方式，无须重新部署新路径，并且满足两个 API 使用者。

Nginx 模块，例如 with-http_sub_module，可用于内容转换。有趣的是，代理级缓存

应用程序可视作是一种智能代理方案。

7.1.3　理解当前代理

在当前应用程序示例中，我们针对 Create User 以及 News 微服务使用了代理设计模式。全部工作是向微服务中使用该模式，除了创建微服务之外，还需要在 Ngnix 中使用代理策略。

下面查看应用程序中的配置内容，其中包含了指向微服务的两个上游内容，以及应用于其上的相关位置。

不难发现，该配置采用了哑代理方案，其原因在于，代理未执行任何数据修改任务，以及包含最小化智能的其他任务。考察下列代码：

```
worker processes 4;

events { worker connections 1024; }

http {
   sendfile on;

   upstream users servers {
      server bookproject usersservice 1:3000;
      server bookproject usersservice 2:3000;
      server bookproject usersservice 3:3000;
      server bookproject usersservice 4:3000;
   }

   upstream orcherstrator servers {
      server bookproject orcherstrator news service 1:5000;
      server bookproject orcherstrator news service 2:5000;
      server bookproject orcherstrator news service 3:5000;
      server bookproject orcherstrator news service 4:5000;
   }

   server {
      listen 80;

      location / {
         proxy pass       http://users servers/;
         proxy redirect   off;
         proxy set header Host $host;
         proxy_set_header X-Real-IP $remote_addr;
```

```
            proxy set header X-Forwarded-For $proxy add x forwarded for;
            proxy set header X-Forwarded-Host $server name;
        }

        location /news/ {
            proxy pass          http://orcherstrator servers/;
            proxy redirect      off;
            proxy set header Host $host;
            proxy set header X-Real-IP $remote addr;
            proxy set header X-Forwarded-For $proxy add x forwarded for;
            proxy set header X-Forwarded-Host $server name;
        }
    }
}
```

上述应用程序使用了两种模式。对于 News 微服务，代码使用了聚合器设计模式；而对于 UsersServices 和 OrchestratorNewsService，则采用了代理设计模式。为了更好地理解如何同时使用这两种模式，考察图 7.2。

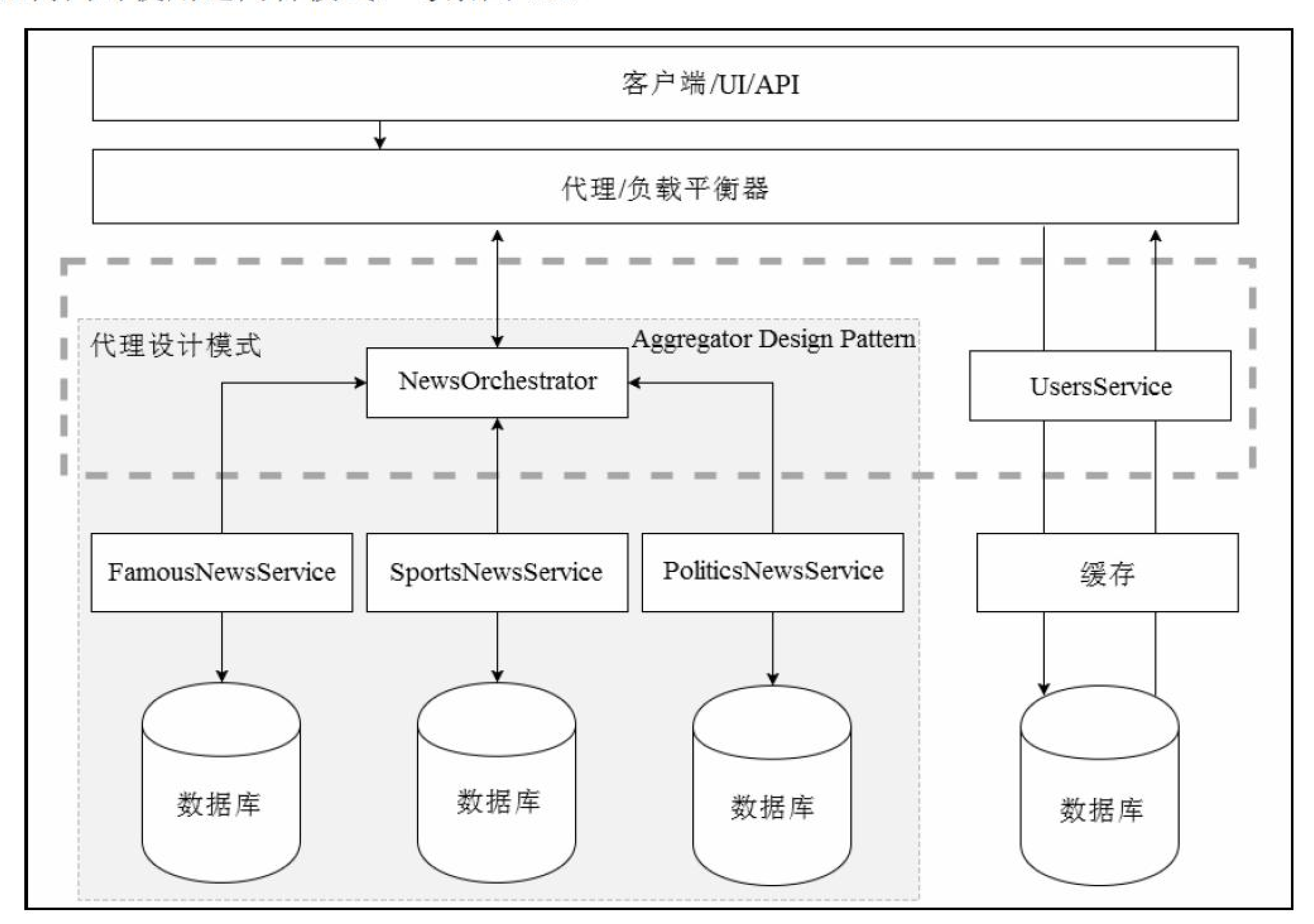


图 7.2

7.2　编排器的代理策略

代理设计模式并未包含复杂的数据编排策略。如前所述，代理仅在请求级别上执行任务。当然，一些源自请求的信息可能会进行较为特殊的处理，但依然无法与之前讨论的编排器设计模式相提并论。

当讨论基于代理设计模式的数据编排时，我们将直接对当前请求进行查看。代理方案并不排斥另一种微服务模式。某些时候，这将有助于将单体应用程序迁移至微服务中。

假设我们有一个要分解的单体应用程序，并将其域转移到微服务中。这一举措值得称赞，但在关闭单体应用程序时，我们不能停止新特性的开发过程。很快，该策略将使用到代理设计模式，并使应用程序处于工作状态。与此同时，我们将业务转移并将新特性应用到微服务中。在微服务稳定过程中，针对响应某个请求的微服务，将利用代理重定向该请求。图 7.3 展示了此处所描述的方案。

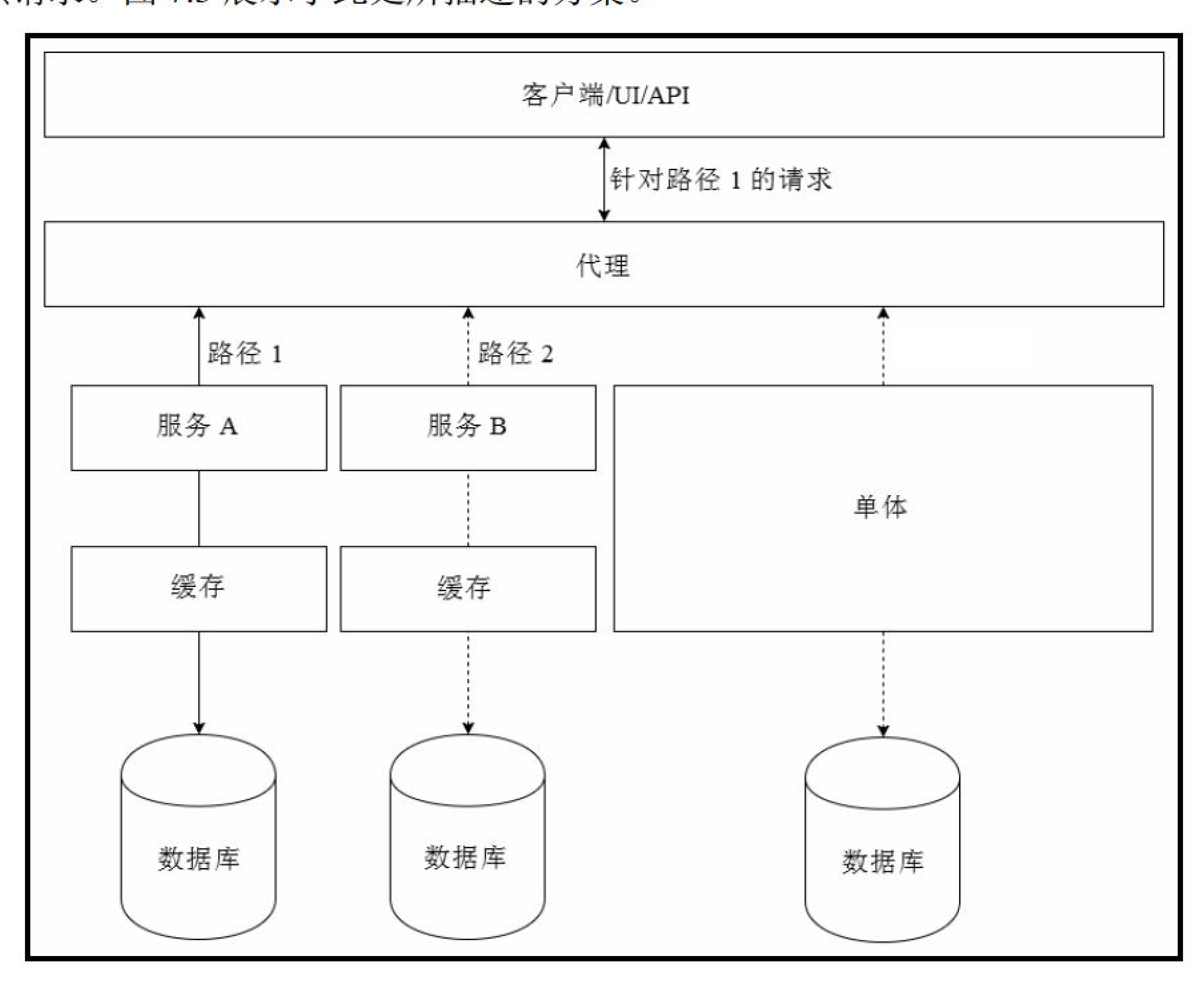


图 7.3

利用该策略，当迁移单体应用程序域并创建微服务时，可将请求重定向至单体应用程序不再存在的地方。

7.3　微服务通信

当谈及代理设计模式时，实际上仅采用 HTTP 协议处理通信问题，其原因在于，该模式一般应用于面向客户端服务层上。

显然，可以在内部服务级别上使用代理设计模式，尤其是利用新的微服务替换某些“贬值”的微服务时。然而，这种应用程序并不常见，在某种程度上，它强制在内部服务层上使用 HTTP 协议。如前所述，这并不是一种最理想的方案。

7.4　模式扩展性

代理设计模式支持 x 轴和 y 轴上的扩展，并引用针对代理访问的、有效服务实例的数量。

根据 Nginx 配置（提供代理角色），当前每项微服务包含 4 个实例，这可以通过对上游配置中微服务实例的引用数量来验证，如下所示：

```
upstream users servers {
   server bookproject usersservice 1:3000;
   server bookproject usersservice 2:3000;
   server bookproject usersservice 3:3000;
   server bookproject usersservice 4:3000;
}

upstream orcherstrator servers {
   server bookproject orcherstrator news service 1:5000;
   server bookproject orcherstrator news service 2:5000;
   server bookproject orcherstrator news service 3:5000;
   server bookproject orcherstrator news service 4:5000;
}
```

在当前应用程序示例中，将采用 x 轴扩展处理扩展性问题——此处仅使用基于代理设计模式的水平扩展。图 7.4 显示了这里所采用的扩展策略。

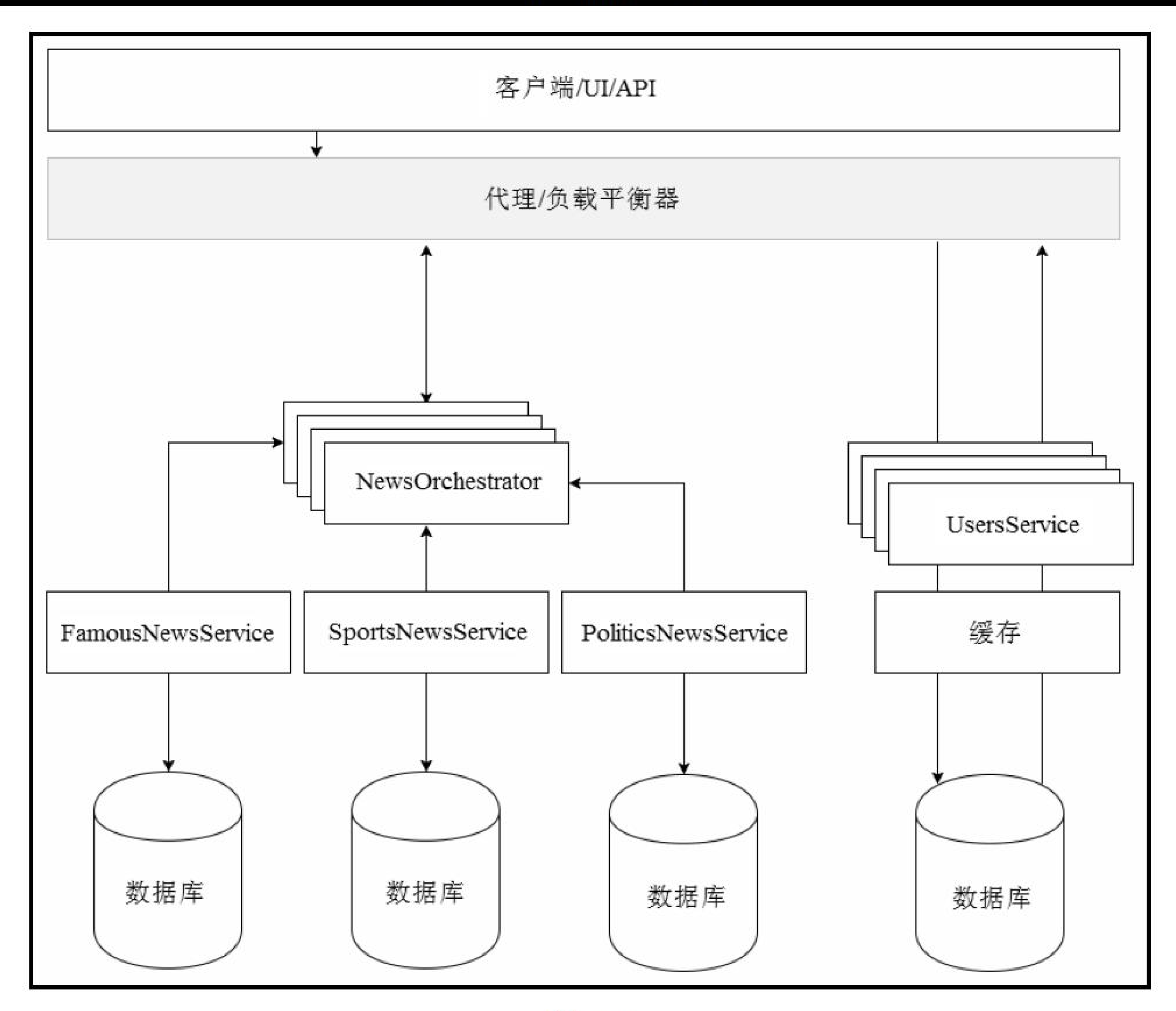


图 7.4

7.5　最佳实践

与其他模式相比，代理设计模式维护和理解起来均相对简单，无论它们是否采用了架构模式。虽然代理设计模式较为简单，但若稍有疏忽，仍会引发某些临界故障点。

这里还需要强调一下，在微服务中发现错误（主要是在架构视图中）并不是一件容易的事情，一些关键问题应引起我们足够的重视。

7.5.1　纯粹的模式

到目前为止，我们在新闻门户网站中使用了两种模式，并可根据需要应用更多的模

式。然而，在某些时候，可能并不一定要使用多个模式。对于当前应用程序上下文，代理设计模式已然足够。

利用前述方案，最理想的方法是保留或尝试保留尽可能多的纯微服务，这意味着微服务自身已可满足要求。当微服务需要利用同步协议与另一个微服务通信时（主要位于内部服务层），那么，该微服务并非是一个纯微服务。

考虑到微服务间内部通信的开销，这一类非纯微服务可导致扩展问题。初看之下，尝试增加接近代理的实例数量似乎是合理的，但问题并不在此。

在处理代理设计模式时，创建能够自给自足的微服务是一种很好的实践方案。也就是说，无须咨询其他微服务即可执行某项任务。

7.5.2　瓶颈问题

代理工具执行的任务相对简单，通常情况下，这种类型的工具表现得非常出色。然而，与任何软件一样，资源并非是无限的。

有时，代理背后的微服务可能出现运行缓慢这一类问题，我们会认为问题出在微服务上，因而尝试增加代理背后的微服务实例数量，并以此对性能问题加以改进。此类行为貌似符合逻辑，但需要注意的是，微服务的全部负载均位于代理上，这有可能成为微服务的瓶颈。

重要的是，需要验证安装代理工具的机器资源是否满足预期的负载。此时，对于理解微服务性能降低这一问题，微服务和代理的性能测试和监视行为将变得十分重要。

7.5.3　代理制的缓存机制

某些代理工具设置了代理级别的缓存能力，对于降低应用程序的压力，这可视作一种较好的策略。显然，代理级别的缓存可能难以控制，但如果谨慎使用，其有效性仍十分明显。

7.5.4　简单的响应

许多代理工具可修改源自 HTTP 请求的 HTTP 响应。这种类型的数据变化可能极具吸引力，但这并非是一种良好的实践方案。

一些开发团队已经严格定义了自身的组织结构。其中，代理隶属于 Ops 或 DevOps。代理级别的更改往往会反映出某种官僚主义作风，任何以不恰当方式对响应进行的修改，都将产生难以检测和修改的附带损害。

对此，较好的方式是在应用程序级别控制响应的内容，而不是将任务托管至代理中。

7.6　代理设计模式的优缺点

尽管我们一再强调，但代理设计模式确实是微服务架构中最为简单、有用的模式之一，其优点包括：

- 应用程序客户端使用数据。
- 实现的简单性。
- 可在代理级别采用较好的编程技术，例如缓存。
- 封装微服务的访问。
- 可控制和转移请求。

但是，代理设计模式也可能会导致一些问题，特别是缺少较好的实践方案时，其中包括：

- 瓶颈问题。
- 响应中不恰当的变化。
- 过载识别中的障碍。

类似于其他设计模式，代理设计模式也包含了自身的优点和缺点，重要的是理解其适用性，并将重点放在良好的实践方案上，以便该模式能够正确地为应用程序工作。

7.7　本 章 小 结

本章内容极具指导意义，并讨论了代理设计模式的工作方式、优点及其可能带来的风险。

第 8 章将继续探讨当前项目，并考察一种更加有效的设计模式。

第 8 章　链式微服务设计模式

第 7 章讨论了代理设计模式的功能和实用性，这可视作一种应用较为广泛的模式，尽管该模式常被一些开发人员无意识地加以使用。除此之外，对于单体应用程序至微服务间的迁移操作，我们还介绍了代理模式所包含的灵活性。本章将探讨链式设计模式，这也是一种十分有用的模式。对于采用了微服务架构的大型应用程序来说，链式模式十分必要。

本章将阐述链式模式的功能、应用时机和场所。此外，我们还将讨论链式设计模式自身的优缺点。

本章主要涉及以下内容：

- ❑ 微服务通信。
- ❑ 模式扩展性。
- ❑ 反模式。
- ❑ 最佳实践方案。

8.1　理 解 模 式

通常，业务中的微服务无法针对应用程序提供完整的解决方案，可能还需要使用基于其他域的编译信息。链式设计模式是为了响应和满足这一需求而开发的，它提供了对应用程序请求的单一响应。

该行为类似于聚合器设计模式，旨在对信息提供单一访问点。然而，针对请求的响应方式则包含了不同的特征。

首先让我们回忆一下聚合器设计模式的工作方式，进而明晰两种模式间的差异。

对于负载平衡器，聚合器仅包含单体访问点，即编排器，负责聚合和组织数据以响应某个特定的请求。

在接收了请求之后，编排器针对微服务（负责形成针对当前请求的响应）评估并触发并发过程。每项微服务执行必要的操作，并向编排器发送响应。

编排器将组织数据，将其序列化至单一响应中，进而发送至应用程序的使用者处。图 8.1 显示了此处描述的处理过程。

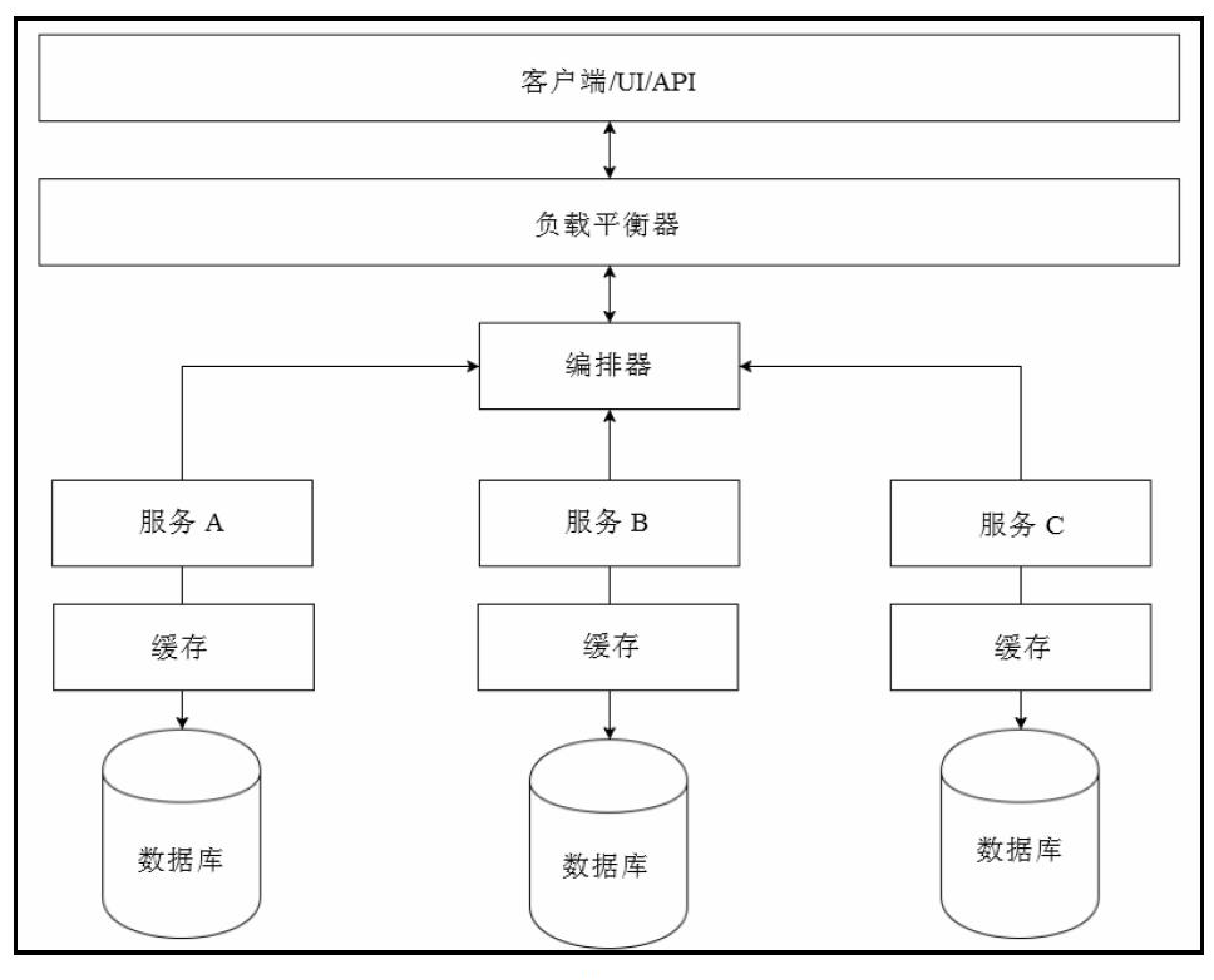


图 8.1

此时，我们可以回忆一下第 6 章中的聚合器微服务设计模式，但本章的目标则是链式设计模式。

由于链式设计模式采用了数据编排设计模型，软件工程师经常说这并不是一个编排模式，而是一个组合数据的模式。在我看来，这两种解释都是正确的。

下面讨论链式设计模式的具体流程，其目标是对通过负载均衡器发送的请求进行响应。与聚合器不同，链式模式针对数据编排器并不包含特定的微服务，因此，应用程序中的任何微服务都可以承担编排响应数据整合结果这一角色。

考察下列情形：服务 A 由负载平衡器调用以响应应用程序的使用者。然而，服务 A 知晓它并未包含完整响应所需的全部信息。在其业务内容中，服务 A 了解到某些信息位于服务 B 中，因而执行针对服务 B 的请求，并请求所需的数据。反过来，服务 B 发现并未包含全部所需的信息，并请求服务 C。服务 C 能够整体响应服务 B 的请求并返回该响应。在接收到服务 B 的响应后，服务 B 形成了当前数据并将其发送至服务 A。随后，服务 A 接收服务 B 的响应，并同样执行数据的合成操作。最终，服务 A 的响应表示为源自

服务 A 自身、服务 B 以及服务 C 的信息合成结果。待全部数据整合至单一响应后，服务 A 将返回应用程序使用的期望值。

数据合成的操作流程可通过图 8.2 进行验证。

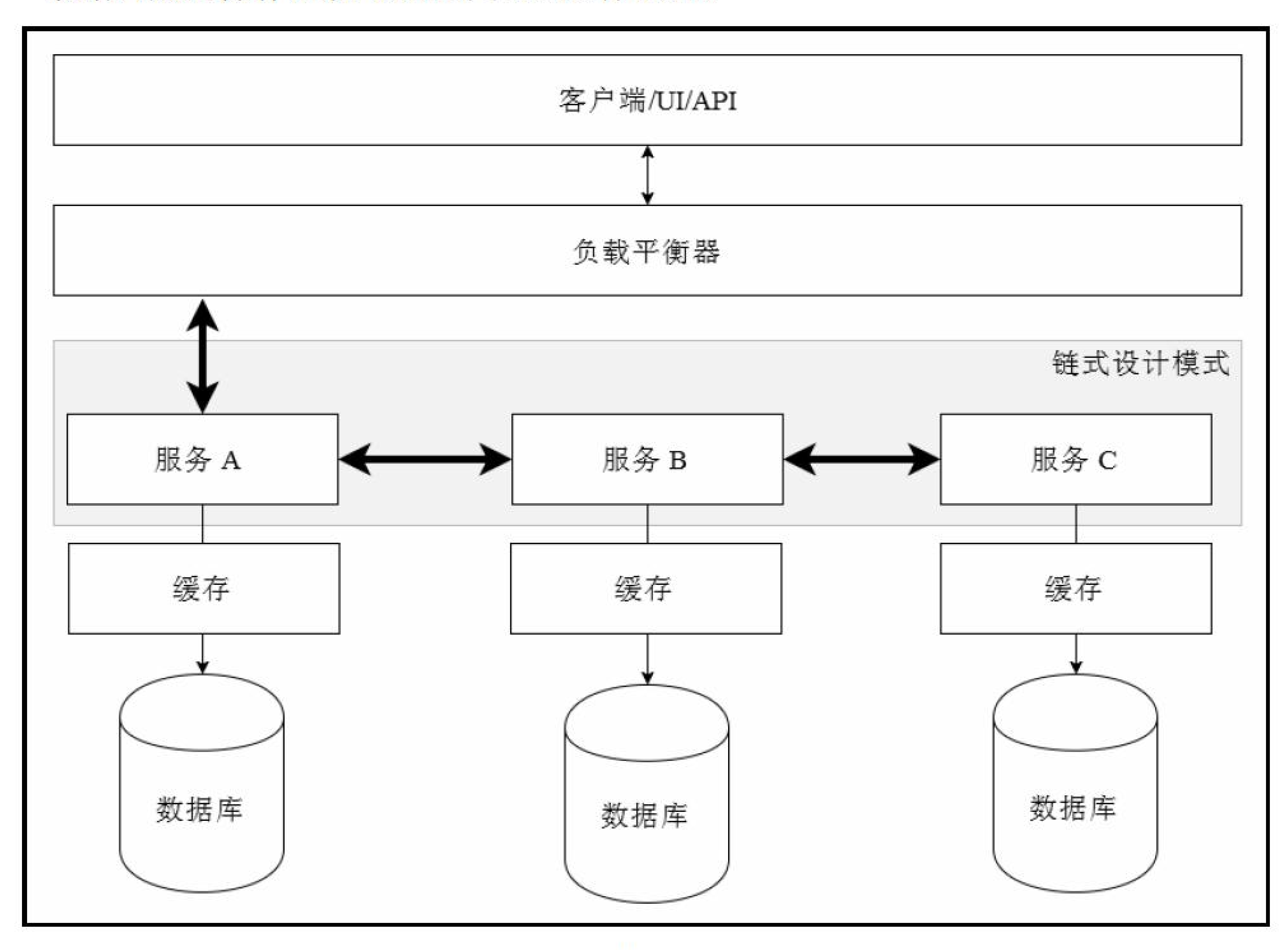


图 8.2

图 8.2 中的流程表示为顺序方式，但实际上，在链式设计模式中，并未对数据的合成顺序进行强制要求。源自负载平衡器中的请求可通过 B→A→C 或 B→C→A 直接发送至服务 B 中。

在当前应用程序中，尚未实现链式设计模式，稍后将对此予以实现。事实上，我们可以在新闻服务中对其加以使用，而不再采用聚合器。这两种模式都解决了向 get_all_news 端点提供公共、统一响应这一问题。如果针对新闻微服务使用了链式模式，则不再持有 OrchestratorNewsService，且需要将 get_all_news 端点传递至应用程序的另一部分中，鉴于当前域特征，这一部分内容似乎缺少一定的语义。因此，针对新闻微服务中的此类业务，聚合器的功能则更加强大。

如果选择了链式设计模式而不是聚合器，那么，图 8.3 显示了当前应用程序的示意图。

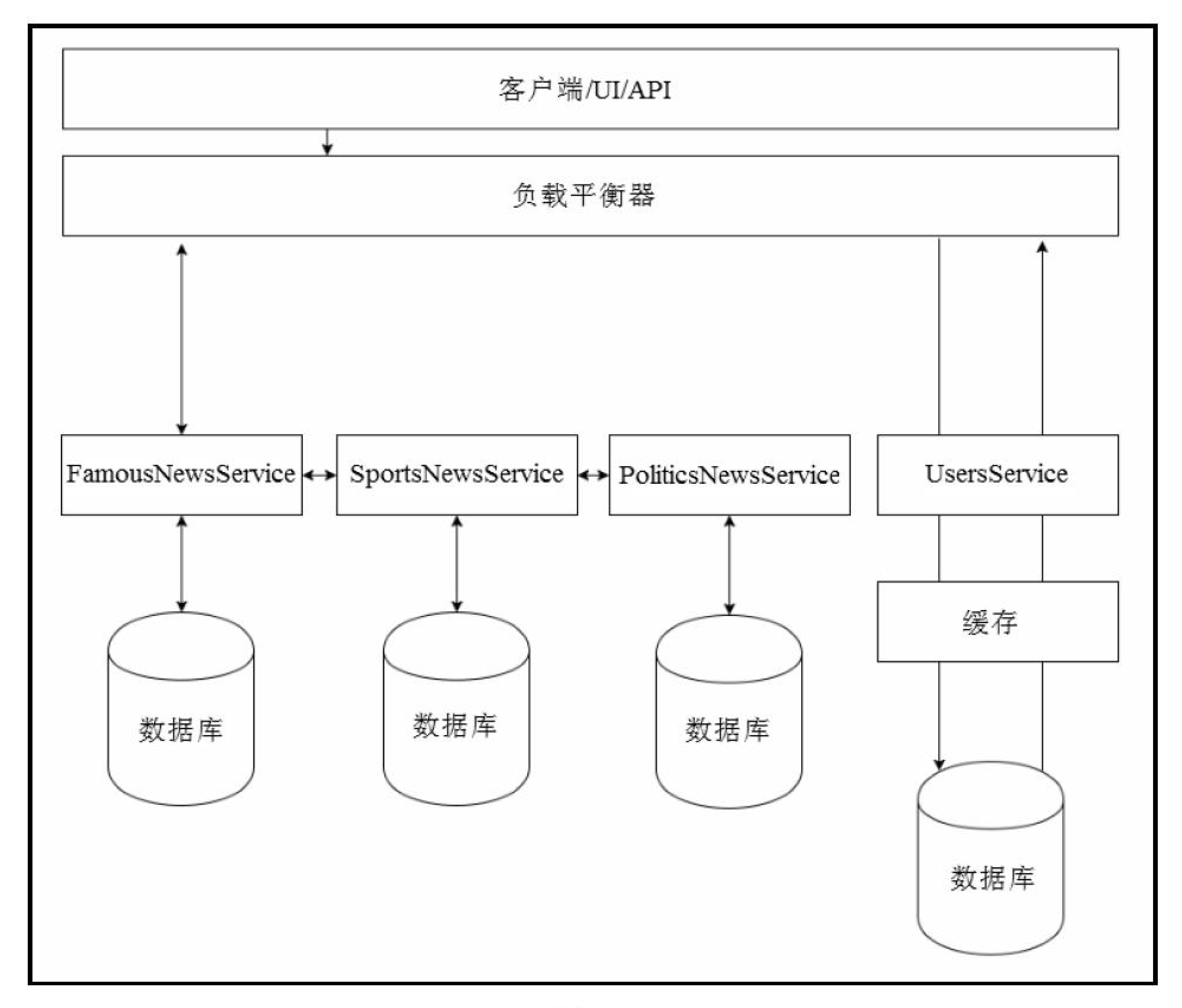


图 8.3

8.2　数据编排和响应整合

基于链式设计模式的数据编排和响应整合包含了一定的复杂度，且分别位于整合处理过程以及调试过程中。

使用链式模式合并数据的过程不会出现在单个点上，因为我们通常习惯于使用聚合器。在链式设计模式下，数据整合以渐进方式进行，并且部分出现于请求链的每个调用阶段。最终结果将在所有请求处理结束时完成。

微服务调用的内部链越长，那么，数据整合过程也就越长。

当采用链式设计模式时，领域驱动设计（DDD）可有效地降低微服务于内部执行的链调用数量。为了不会对应用程序使用者产生负面影响，不应包含微服务间较长的通信

链。需要注意的是，此处为微服务间的直接通信模式，因此，我们不仅向应用程序的使用者传递每项微服务的处理时间，还包括微服务间通信的延迟时间。针对于此，较长的通信链除了降低速度之外，还会对调试过程和数据的维护产生负面影响。

8.3　微服务通信

如前所述，微服务架构中存在两种通信模型，即同步模式和异步模型。当使用链式设计模式时，推荐使用同步通信模型，其原因在于，应用程序的使用者正在等待一个完整的响应，该响应由来自一个或多个微服务的数据组成。如果通信模型非同步，响应的整合控制可能会包含一个回调系统，该回调系统的复杂性会增加并降低可扩展性。

重要的是，利用同步通信模型，我们将生成一个阻塞通信模型。这里，阻塞通信模型是指，应用程序的使用者等待微服务间通信链中被整合的响应结果。微服务间的通信链越长，应用程序使用者等待的时间也就越长。

利用同步通信模型，最简单的实现是使用 HTTP（超文本传输协议）。某些开发人员则更喜欢采用另一种同步通信类型，例如二进制通信协议，或者简单地使用源自发送数据包的压缩程序。作为 HTTP 的替代方案，如 Thrift、Avro 和 gRPC 等工具，它们都有一个二进制序列化器和一个同步信息发送器；而 MessagePack 和 Protocol buffer 仅执行数据包序列化操作。

截至目前，我们所讨论的工具均不含自身的优缺点。当确定采用哪一种工具时，通常较为有效的方式是进行性能测试，进而理解可扩展模式及其相关行为。

8.4　模式扩展性

链式设计模式支持 y 轴、x 轴以及 z 轴模型中的扩展，且均与代理访问以重定向请求时的、有效的服务实例数量有关。

一种较为常见的做法是使用基于链式设计模式的代理设计模式，相应地，代理负责标示微服务；据此，链式模式将启用微服务间的通信，进而整合某个响应。

另一个较为常见的实践方案是，当采用链式设计模式时，每项微服务均不含自身的服务器，如 Nginx，这与之前采用的方法稍有不同，因而包含了一个应用程序层，并需要对相关实例进行操控。

图 8.4 展示了常见的扩展模型。其中，在对通信链中较为缓慢的微服务进行识别后，

将创建经标识后的、新的微服务实例，并以此对性能加以改善。图 8.4 中，我们选择通过 *x* 轴或垂直扩展模型创建更多的服务 B 实例。

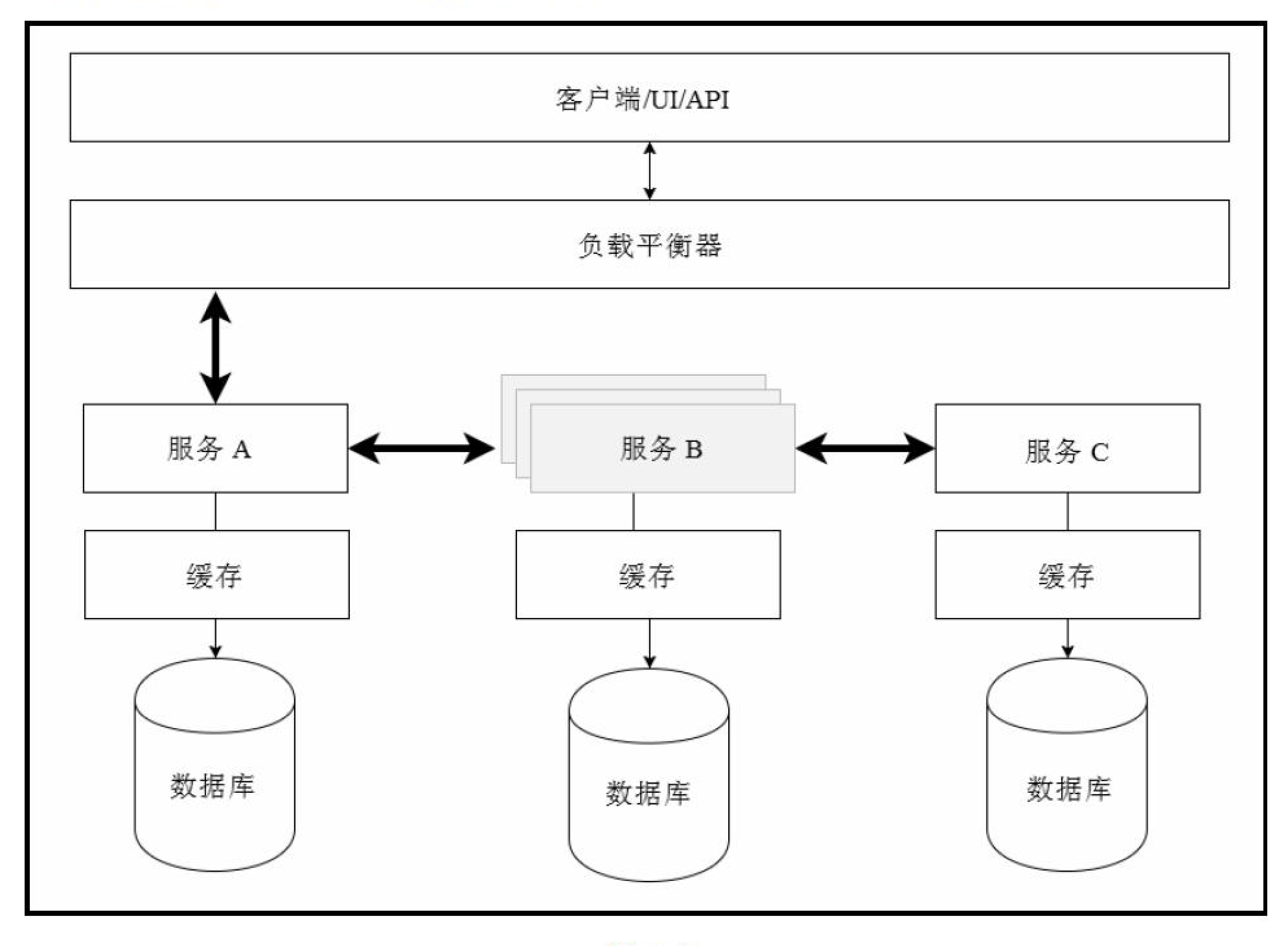


图 8.4

可以使用链式设计模式将可伸缩性策略应用到应用程序的各个部分，但在这种模式的可扩展性方面，有一点可能会受到影响，即微服务间的直接通信层。此时，较好的选择方案是尝试最小化微服务间的直接通信，对此，DDD 以及异步技术可有效地降低调用次数。

8.5　“大泥球”反模式

所有科幻迷都应该知道或听说过死亡之星（Death Star）。在微服务体系结构中，“死亡之星”用于描述“大泥球”（Big Ball of Mud）这一反模式所导致的问题，特别是当我们使用链式设计模式时。

对于域中缺乏良好定义的微服务，这往往会产生“大泥球”反模式，使得微服务间彼此依赖以实现某些琐碎的任务。这类错误将在微服务间产生一系列不必要的调用，并

导致复杂的修正问题，例如延迟现象，严重时还会出现循环部署依赖关系。

图 8.5 显示了“大泥球”反模式的工作示意图。通过观察可知，图 8.5 中的通信模式类似于“死亡之星”。

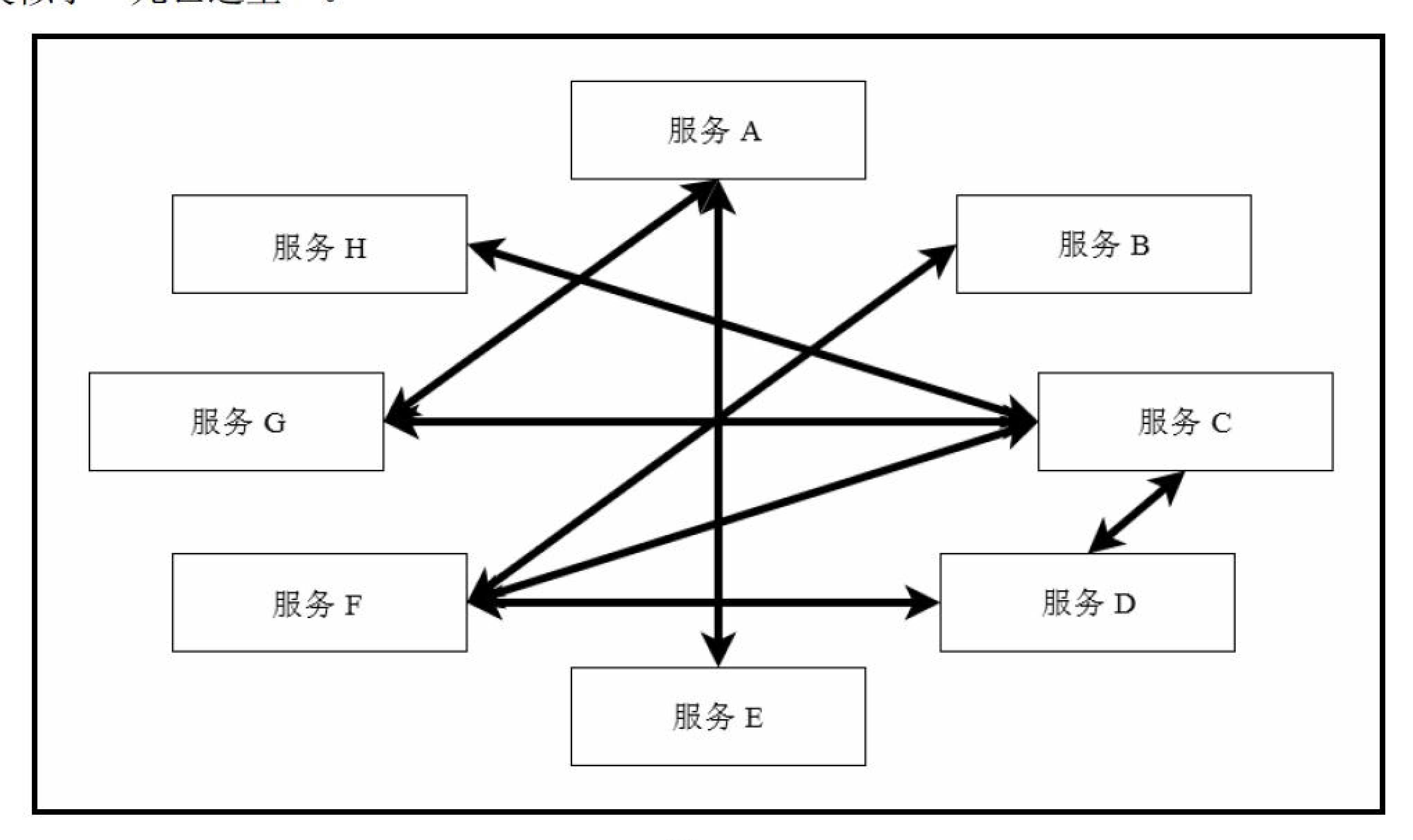


图 8.5

对于某些微服务时，自由通信似乎较为有趣，但这也表明域定义尚不够充分；而直接循环通信依赖则说明微服务包含了较强的耦合，并强制在发布过程中予以干涉，以避免不可用性的出现。

下列内容显示了“大泥球”反模式的主要特征，以供读者加以甄别：

- ❑ 域缺乏良好的定义，并强制与其他微服务进行直接连接。微服务的访问点不需要足够的数据处理某项任务，这将在其他微服务中进行搜索，并强制执行数据干涉。
- ❑ 强制性的直接通信。当大多数任务或全部任务都必须进行微服务之间的直接通信时，这将会导致以下问题：微服务处于“贫血”状态，或者应用程序整体上有所欠缺。
- ❑ 集群部署。当某项微服务因为需要另一个微服务而无法发送至产品中时，或者，当微服务内部签名变化导致其他微服务崩溃时，便会出现“死亡之星”这一类反模式问题。在微服务间构建业务依赖关系意味着，处理过程无法自动、流畅地执行。

如果应用程序符合上述某项特征，那么，该程序正在朝着“死亡之星”的方向发展；如果应用程序与上述多项特征吻合，那么你简直就是帕尔帕廷皇帝[①]！

8.6　最佳实践方案

链式设计模式易于实现，特别是处理我们已经习惯的工具，如 HTTP 协议。然而，考虑到微服务间直接通信过渡使用所导致的难以解决的问题，链式设计模式往往维护起来较为复杂。

在使用复杂模式的应用中，例如链式设计模式这一类场合，一致的日志操作将有助于识别异常情况。但在分布式通信应用程序中，这实现起来较为困难。

为了有助于识别微服务间通信流中可能出现的错误问题，我们可以采用相关 ID 这一方式。

相关 ID 可帮助我们获得分布于多项微服务间的整体任务概况。一种相对简单的、基于 HTTP 的相关 ID 实现方法是发送请求头中的 UUID，并将该 UUID 用作标识符并写入日志。

从之前所讨论的各种模式来看，这一点尤其值得我们注意。当使用链式模式时，稍有疏忽即会对开发路径产生破坏，而调整过程也会变得异常复杂。相比之下，一些最佳实践方案可降低问题出现的几率。

8.6.1　纯微服务

在业务设计过程中，应采用纯微服务，这意味着，微服务必须在其域中是非常小的，并且完全能够在不受其他微服务干扰的情况下执行其功能。

8.6.2　请求一致性数据

考虑到降低与微服务使用者之间的冲突，开发人员通常会制定相关策略，以对数据进行推断。当然，我们也无须对这一类冲突产生恐惧心理。在其端点处，微服务必须接收完成创建任务所需的所有信息。

当采用缺乏一致性或者糟糕的数据时，尝试与微服务协同工作就像是坐进出租车，并让司机去猜测目的地一样。

[①] 帕尔帕廷皇帝是电影《星球大战》中的头号反派角色——译者注。

8.6.3　深入理解链式设计模式

如前所述，某些时候，微服务间的直接通信不可避免。此时，链式设计模式可发挥其功效。然而，较长的通信链同时也意味着维护工作将变得更加复杂。

必要时，直接通信的维护工作可在单级上进行，如服务 A→服务 B。若直接通信包含两级或多级，如服务 A→服务 B→服务 C，那么，系统故障的风险、部署的复杂性和可扩展性都在不断增长。

若多级微服务之间存在直接通信，其实现过程会创建一个系统故障点，特别是执行直接通信并为终端用户收集数据时。

微服务间的直接通信并不会被禁止，但应尽量避免使用。

8.6.4　关注通信层

由于链式设计模式采用了直接通信，因而应尽可能地降低此类通信产生的延迟。注意，当处理通信链和微服务时，应用程序使用者将等待请求的响应结果。

为了减少微服务间与直接通信相关的延迟量，建议使用二进制协议工具或者在使用协议时进行调试。

8.7　链式设计模式的优缺点

与其他模式一样，链式设计模式包含了自身的优缺点。但是，若未经良好实现，链式模式会引发更加复杂的问题。

下列内容列举了链式模式的一些优点：

- 实现过程更具实践性。
- 业务推动力。
- 独立的可伸缩性。
- 微服务访问的封装。

尽管如此，该模式的一些缺点也应引起我们的注意，其中包括：

- 延迟问题。
- 数据持有者往往难以辨识。
- 调试的困难性。

因此，与微服务协同工作并非易事，除了自身技术之外，诸多因素均会对其产生影响。在后续章节中，我们将继续探讨如何在应用程序中正确地使用微服务，并理解实际

操作过程中的最佳应用方式。

8.8 本章小结

本章介绍了链式设计模式的操作过程，包括其应用方式，以及如何帮助我们处理某些通信问题。

第 9 章将继续对当前项目加以讨论，并学习一种新的设计模式。

第 9 章　分支微服务设计模式

前述章节讨论了聚合器设计模式以及链式设计模式的协同工作方式。本章将介绍一种名为分支设计模式的操作，该模式可视为聚合器模式和链式模式间的一种变化版本。

分支设计模式可视作聚合器和链式设计模式的进化结果，以更好地服务于应用程序的业务层。

在本章的结尾，我们将能够识别、分类和理解该模式的特征，并发现分支设计模式的最佳应用场合及其概念规则。

本章主要涉及以下内容：

- 数据编排。
- 微服务通信。
- 模式扩展性。
- 最佳实践方案。

9.1　理解模式

前述章节讨论了一些模式的应用方式，不难发现，每种模式均包含特定的用途。从这个意义上讲，一些模式对技术功能提供了较好的支持，如可扩展性、实用性以及弹性。除此之外，其他一些模式则更关注于业务层。

当我们比较聚合器设计模式和链式设计模式时，就会出现这种情况：两种模式旨在对业务加以改善，但很明显，聚合器设计模式更多注重于技术方面；而链式设计模式则寻找相应的解决方案，以对业务提供相关服务。在某些情况下，对于应用程序来说，此类解决方案可能并不是最佳结果。

分支设计模式则是聚合器设计模式的扩展，并支持两个微服务链中的同步响应处理。对于技术内容与业务层间所产生的冲突，分支模式试图寻找一种中间方案，其目标是在单一模式中整合聚合器设计模式和链式设计模式的优点。

如前所述，链式设计模式的缺点在于，对于数据整合或任务执行，相关调用位于较长的链中；而分支模式则采用聚合器降低链的尺寸，同时在微服务调用中构建一种并发机制。

初看之下，分支模式这一概念较为复杂。下面将对问题的每一部分加以讨论，以使相关解决方案趋于完美。

当采用链式设计模式时，常会生成较长的同步通信链，使得应用程序的使用者等待构成该决策链的、每项微服务的执行操作，如图 9.1 所示。

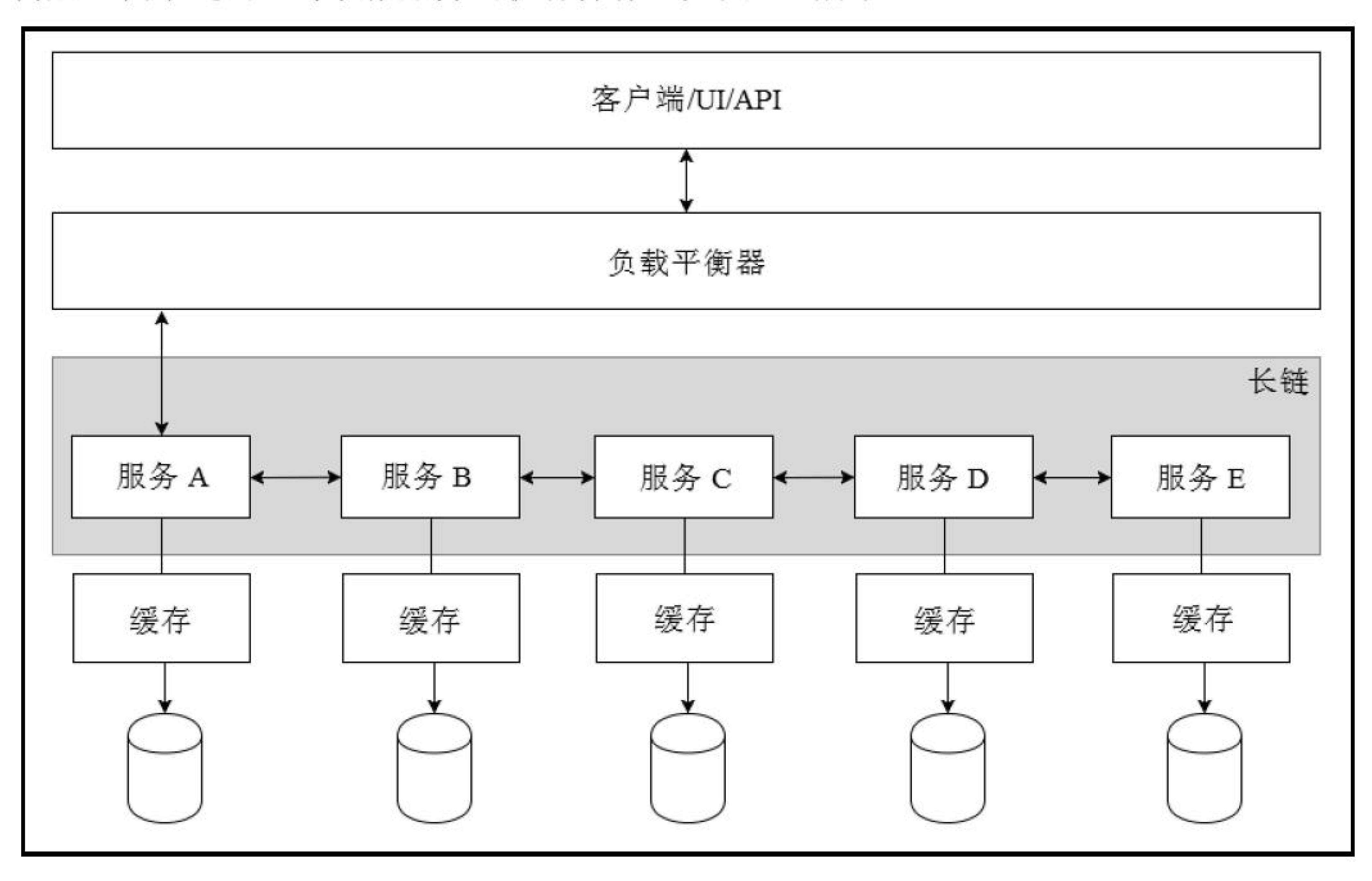


图 9.1

在微服务调用中，避免此类长链类型似乎较为简单，但事实并非如此。具体来说，任意创建同步调用是一种较为常见的做法。在图 9.1 中，我们很容易看到长链的存在；但在实际操作中，通常会涉及数百、数千个端点，长链往往难以发现。

图 9.2 显示了 3 个发送至应用程序的请求。其中，每个请求均包含了自身的执行链。因此，识别每个同步调用链及其尺寸将会是一项越来越复杂的任务。

针对理解过程、维护操作、向当前业务添加值以及技术内容，为了降低其复杂度，我们可以采用分支设计模式。

分支设计模式涵盖了以下规则：

- ❑ 采用直接调用链的响应整合结果，无法将一个直接调用扩展到另一个微服务上。
- ❑ 如果响应需要更多的数值，可创建一个聚合逻辑，并根据需要对多个链触发并

行请求。

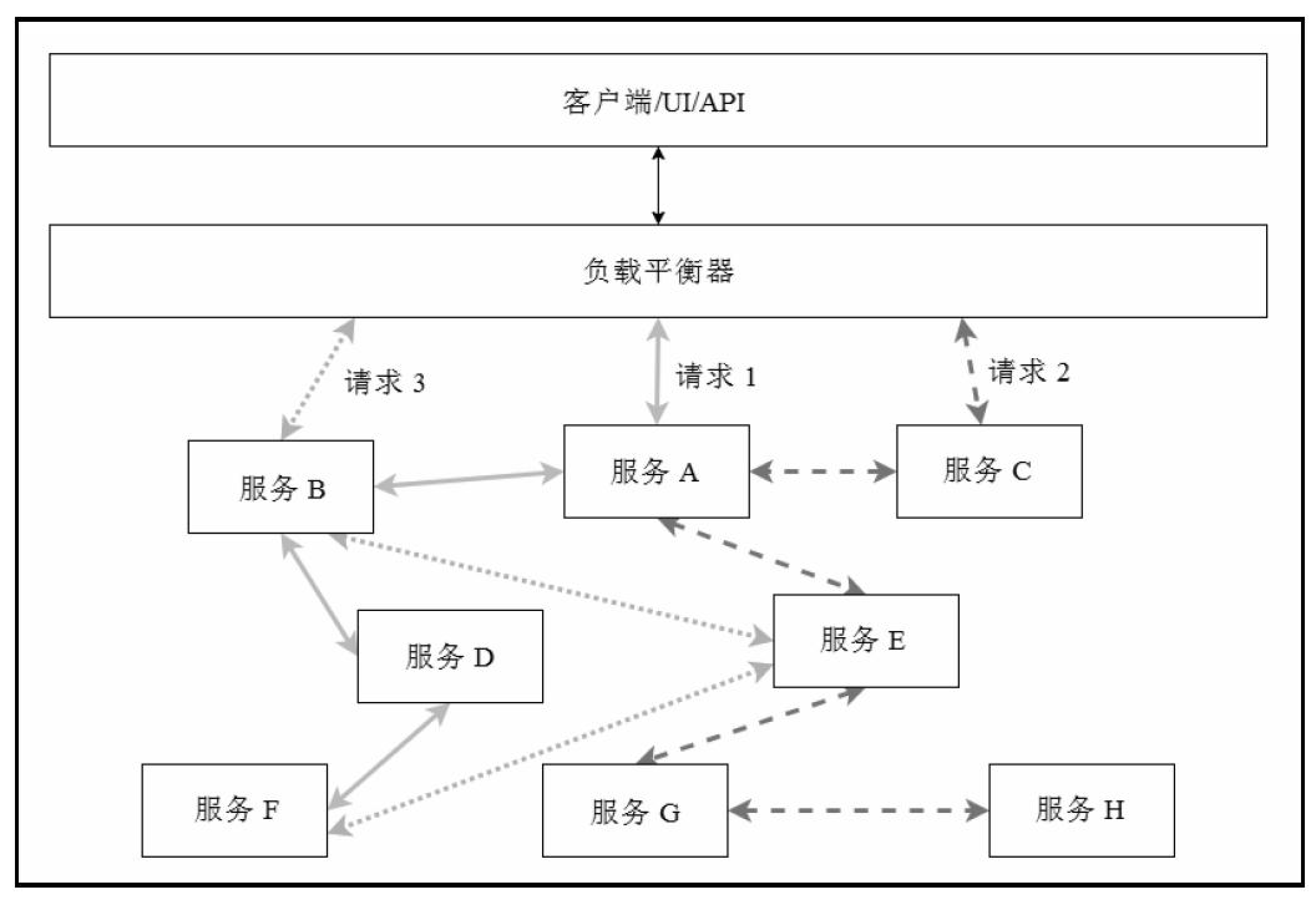


图 9.2

让我们更好地理解这一过程。从技术角度来看，存在一个微服务负责编排数据（该数据置于应用程序中的所有其他微服务之上），以及一个聚合器设计模式。差异主要来自编排器下面的微服务的行为。

而在聚合器中，编排器将向微服务传播消息；在分支设计模式中，编排器向微服务和微服务链发送消息。其中，这一类链限定于一个同步调用中。

分支设计模式的基本理念是支持微服务间的直接同步通信，但会降低出现于应用程序内部的长请求链的开销。同样，每个微服务域对于维护适当的调用结构是至关重要的。

图 9.3 展示了分支设计模式的行为。图中，前端负责生成请求，并传递至负载平衡器；同时，该请求根据负责编排数据的端点而分布。每个编排器将咨询构成当前信息所需的微服务，这些微服务可能呈现为个体状态，或者是微服务链。对于内部层的请求、并发微服务，以及较短的同步直接调用链，应用程序的整体响应时间将大幅减少。

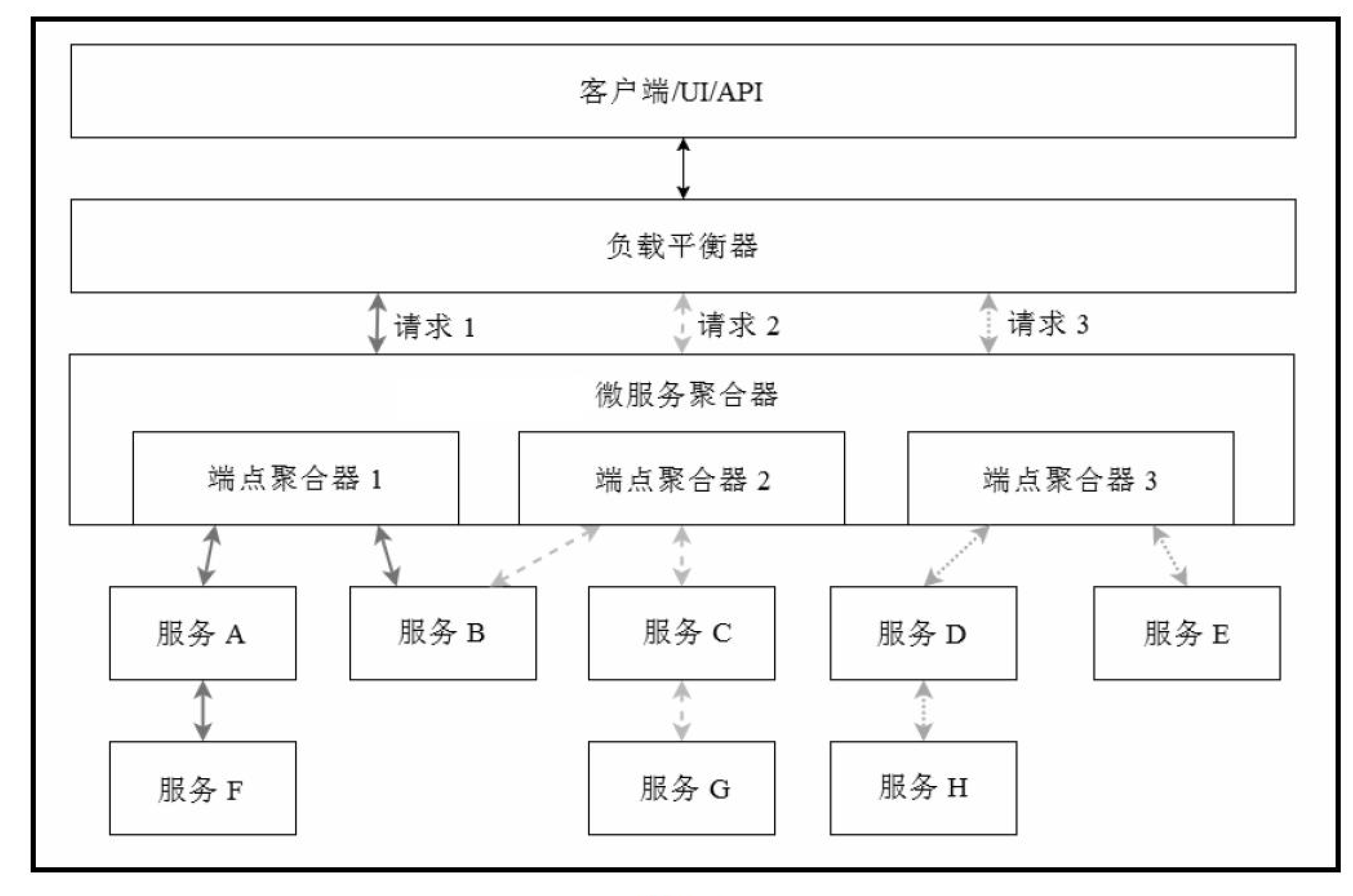


图 9.3

9.2　数据编排和响应整合

对于分支设计模式的数据整合，其复杂度一般认为是最高的。这里，高复杂度源自以下事实：当采用分支时，我们通过交替方式或同步方式，同时使用了数据的整合操作以及数据的编排操作。

当采用分支设计模式时，较为常见的情形是，数据的整合出现于微服务的内部层中。显然，数据编排则位于面向公共层中。读者需要牢记的是，复杂的数据搜索意味着更多的时间和机器资源来创建完整的响应。

图 9.4 显示了数据合成在内部层中的执行处理过程，以及面向公共层的数据编排过程。下面让我们进一步查看图中的执行流程。

（1）微服务聚合器接收来自 API 的请求，并将响应整合至应用程序的使用者处。

（2）编排处理将执行随后的步骤，并于其中识别数据搜索所需执行的命令。

（3）将会触发 3 条命令，这些命令知晓在哪些微服务中获取数据。

（4）当前，内部层微服务开始工作，并发送命令所请求的信息。

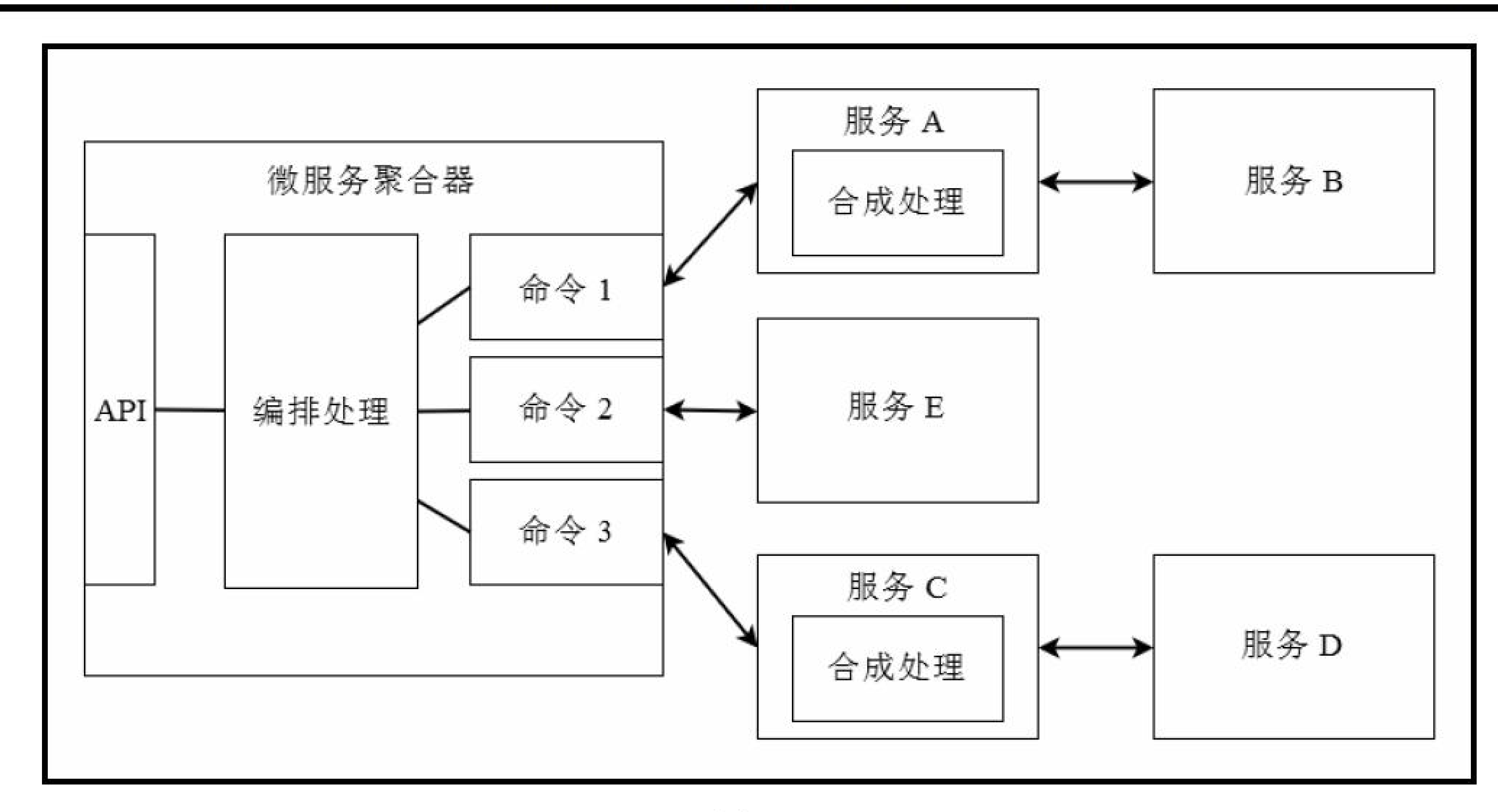


图 9.4

（5）一些微服务，如服务 A 和服务 C，确定它们没有能力完成所请求的任务，并分别调用服务 B 和服务 D。

（6）服务 B 和服务 D 分别针对服务 A 和服务 C 以同步处理方式返回数据。

（7）通过来自服务 B 和服务 D 的数据，服务 A 和服务 C 开始合成数据，并向发起通信的命令返回单个响应。

（8）相关命令将所接收的数据返回至编排处理过程，依次处理信息并整合数据。

（9）最后一步是将整合后的数据返回至应用程序的使用者。

上述流程涉及较多内容且相对复杂，但并不意味着其过程缓慢。显然，诸如延迟和性能问题将会始终萦绕于使用微服务架构的开发人员的脑海中。微服务模式自身并不会减缓应用程序的运行速度，一些导致速度缓慢的因素包括通信层、域定义以及较差的扩展实践方案。

9.3　微服务通信

与链式设计模式类似，分支设计模式也采用了微服务间的同步通信模型，因而可借鉴某些同步通信执行方式。

第一种方法是采用某种协议或者直接消息的同步通信，这也是微服务间同步通信最为简单的使用方式。HTTP 则是较为著名一个协议示例；而作为直接消息，我们可使用远

程过程调用（RPC）。但是，分支设计模式也包含自身的特性——该模式仅与编排和数据合成过程协同工作。

针对数据合成，协议的使用则较为简单且广泛被人们所接受。不同的是，当涉及编排处理过程时，则无法采用相同方式予以处理，其原因在于，数据的编排出现于相同的微服务中。尽管如此，微服务中的内部通信仍可通过下列 3 种方式工作：

- ❑ 顺序方式，即不存在并发或并行机制。这意味着，当需要将命令消息发送至微服务时，全部处理可视作一个序列。如果需要执行 4 条命令构成数据，那么所有的命令都将按顺序执行。
- ❑ 线程。其中，可以使用 POSIX 线程（pthreads）和绿色线程。对于开发人员来说，线程控制并非易事。如果某个线程失效，数据编排将会受到损害。但是，线程也是最为实用的处理方式——不需要使用到编程语言的外部组件在命令的执行过程中创建某种级别的竞争或并行机制。
- ❑ 消息代理。使用事务消息代理在微服务中传输敏感数据是一种很常见的做法。消息代理的缺点是在微服务中添加了物理组件。相比之下，其优点则是能够执行数据传输过程中更具弹性的策略。与事务协同工作这一简单的事实已然可视作是一种很好的资源。

图 9.5 显示了微服务中消息代理的内部工作方式。不难发现，消息代理在编排处理和负责调用每个微服务或微服务链的命令之间建立了一个桥梁，以便为应用程序使用者创建唯一的响应。

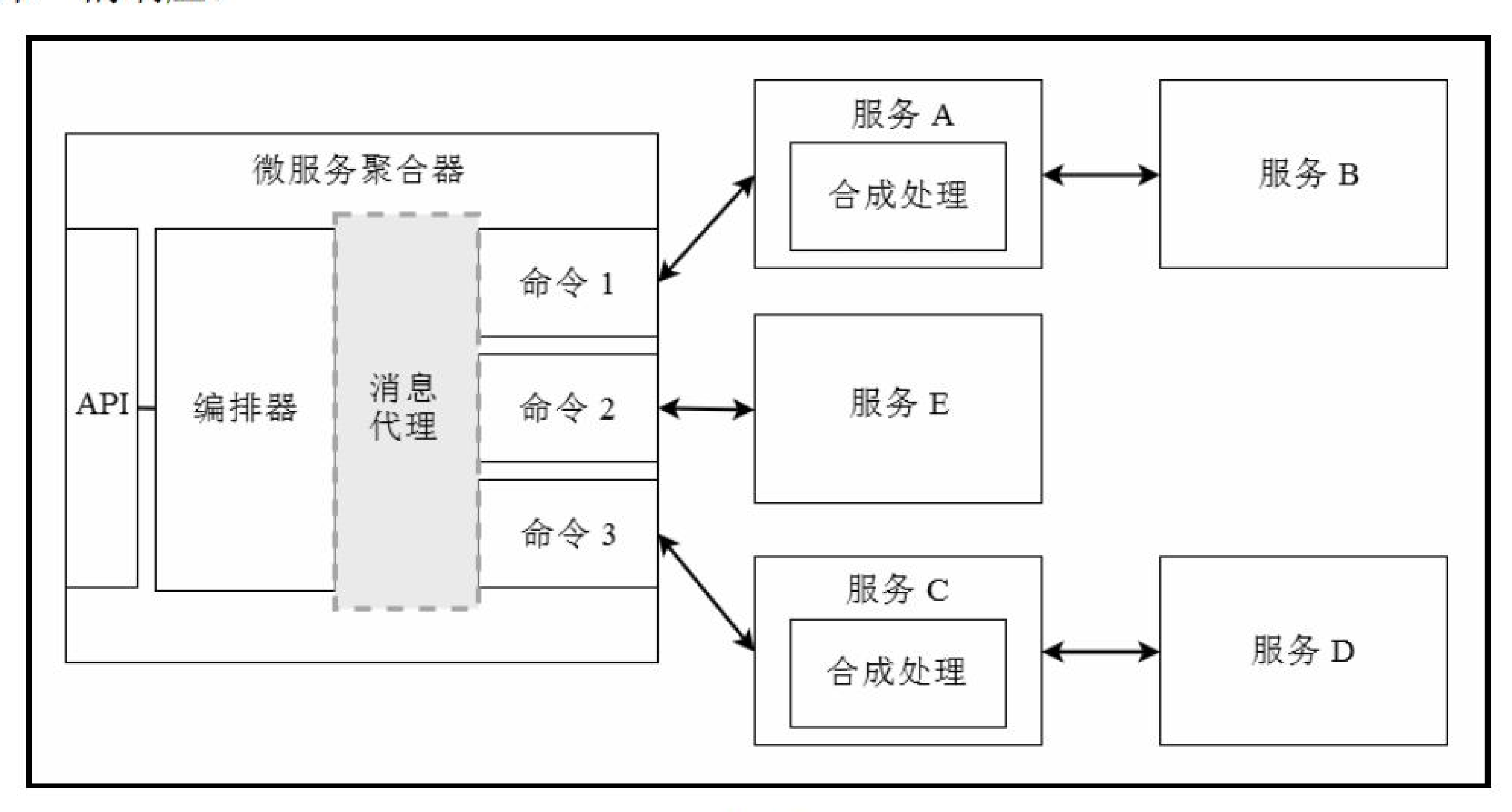


图 9.5

正如在微服务内部使用消息代理一样，我们可以在编排微服务和其他微服务之间使用非常相似的通信策略。

通过将消息代理用作通信层，甚至采用链式通信的微服务也开始使用这一物理组件。图 9.6 显示了之前所描述的概念。

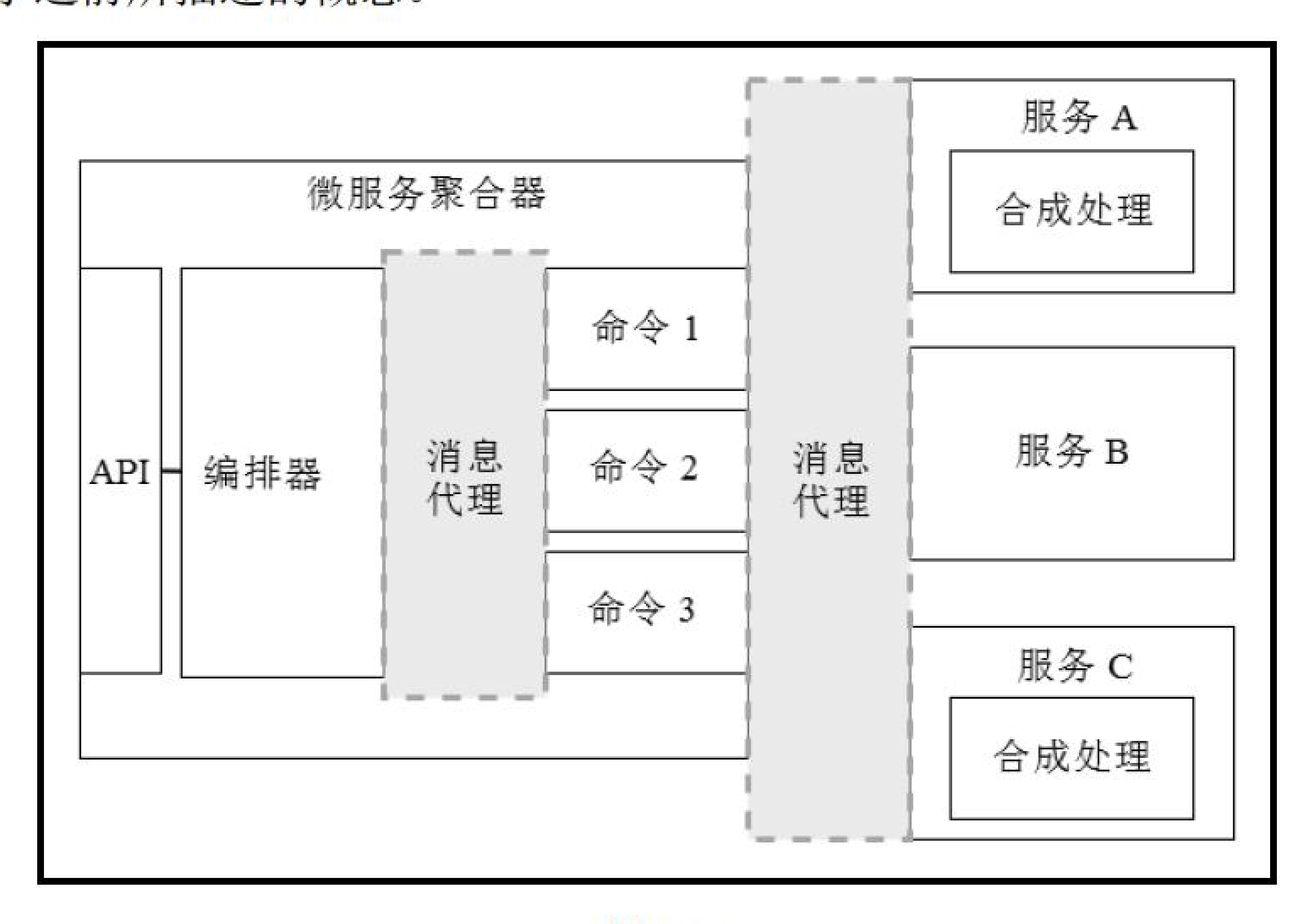


图 9.6

9.4　模式扩展

分支设计模式可视为链式模式和聚合器模式的组合结果。因此，其中的所有扩展和应用模型均适用于分支设计模式，但某些地方仍需要引起我们足够的重视。

根据扩展立方体模型，我们可使用 x 轴、y 轴和 z 轴扩展。然而，对于当前模式的每一部分，轴向的使用方式也有所不同。

对于微服务的编排行为，仅可使用 y 轴和 x 轴，这一限制条件主要体现在：除了与之通信的其他微服务发送的数据之外，编排器不具备任何数据访问权限。

当谈论负责数据操控的微服务时，情况则有所变化。此时，全部轴向均可针对应用程序提供可扩展性。

需要注意的是，如果工程团队选择将物理组件用作微服务间的通信层，而非协议，那么，该物理组件同样需要包含可扩展性，或者至少采用一种弹性策略，以保证应用程序不会崩溃。诸如物理组件集群，甚至是简单的主/从策略都可使不可用现象显著降低。

9.5　最佳实践方案

就具体实现来说，分支设计模式是最为复杂的模式之一，对于维护工作尤其如此。针对这种模式类型，遵循良好的实现方案往往具有强制意味（而不再是一种建议）。

针对这一主题，我们将讨论采用哪些较好的实践方案，以在使用分支设计模式时避免出现问题。

9.5.1　域定义

链式设计模式可视为一种标准，并支持较低的域定义级别——顺序调用长链即体现了这一点。然而，分支设计模式却不存在这种错误类型的空间。

当我们将分支视为一种应用程序模式时，设置较短的调用链则是一种业务选择方案。

在实现分支设计模式之前、期间和之后，可重新访问应用程序的域，并采用 DDD 处理过程适当地限制每个微服务的范围。

9.5.2　遵守规则

在决定采用分支设计模式之前，应全面理解该模式的操作行为。分支设计模式是本书最复杂的模式之一，因此，完整地理解该模式可视作最基本的要求。

此处指的是模式的实现规则。创建混合或不完全符合模式的内容意味着性能的损失、维护的极度困难、测试的困难以及许多其他可能带来伤害的因素。

如果某些模式规则因为域业务而需要修改，那么请停下来对此加以思考。这里存在两种可能性：域未经良好定义；或者模式不适于当前业务。

9.5.3　关注物理组件

当使用分支设计模式时，其中涉及两个重要概念，即数据编排和数据合成。将每种抽象概念置于各自的市场地位中并不是一件易事；另外，构建编排和合成操作间的通信机制则更加困难。

通常情况下，为了降低通信的复杂度，可使用消息代理这一类物理组件。针对此类组件，由于应用程序的所有流量和数据都途径此类组件，所以我们必须加倍关注。

提供可伸缩性、实时更新和物理组件的监视行为，对于应用程序的可用性至关重要。

9.5.4 简化行为

模式越复杂，其实现就越简单。丰富的变化和健壮的堆栈进一步增加了维护、理解和学习应用程序的成本。

在分支设计模式中，维护的简单性与降低实现新特性的成本之间具有直接关系。显然，降低复杂度并不意味着域业务不可使用该模式。这与决策相关，且无须考虑未经适当验证这种可能性。

9.6 分支设计模式的优缺点

分支设计模式相对复杂，但同时也十分有效且兼具灵活性。理解相关操作以及应用该模式的时机可有效地防止问题的出现。

下列内容列举了该模式的一些优点：

- 实现的灵活性。
- 独立的可扩展性。
- 微服务访问的封装。
- 合成能力和编排操作。

当然，我们也应该了解该模式的一些缺点，其中包括：

- 可能导致的延迟问题。
- 难以辨识数据的持有者。
- 调试难度加大。

在开始阶段，分支设计模式的复杂性可能令人望而却步。然而，从技术角度和业务角度来看，分支的各种可能性使其成为值得全力关注的模式。

9.7 本 章 小 结

本章讨论了最为复杂的模式之一，即分支设计模式，并介绍了该模式的应用方式和时机，以及实现该模式时所需注意的问题。

第 10 章将继续讨论微服务架构的最新模式。

第 10 章　异步消息微服务

第 9 章讨论了分支设计模式，该模式常与微服务间的直接顺序通信协同工作。本章将介绍并使用异步消息设计模式。对此，我们需要创建另一个微服务，即 RecommendationService，并负责显示应用程序注册用户感兴趣的新闻类型。

本章将实现全部的、与异步消息设计模式相关的概念。最后，我们还将探讨其应用位置和时机。

本章主要涉及以下内容：

- 域定义。
- 数据定义。
- 使用消息代理。
- 模式的可扩展性。
- 反模式。
- 最佳实践方案。

10.1　理解当前模式

我们已经了解到 REST（表述性状态转移）的简单性和实用性，主要原因在于 REST 在市场中的成熟度。另外，REST 层的工具和框架数量对其流行程度也提供了辅助作用。

REST 调用包含一个同步字符，由于当前技术所用的请求/响应模型，同步调用将被阻塞。虽然缺少显著特征，但此类调用类型对于应用程序业务来说通常不可或缺。这也表明，应用程序微服务间存在某种程度的耦合。

纯微服务必须完全能够执行分配给它的任务，而不需要与另一个微服务进行通信。纯微服务的另一个特点是，它执行任务时不需要返回响应，仅是简单地接收请求并执行所需操作。从这个意义上来讲，异步消息传递设计模式可视为一个模型，并可自然地与纯微服务协同工作。鉴于该模式的特性，消息被发送后，不存在针对其他微服务的信息响应或传播。因此，请求生成器无须被告知如何持续工作，它仅发送一条消息并知晓具体位置；某个微服务将获得一条消息并执行所确定的内容。

这种模式无疑是最具可扩展性的。考虑到微服务的同步字符，在任务完成过程中，

将不会对应用程序的最终使用者带来任何不便。

异步消息设计模式可与其他模式结合使用。除此之外，对于微服务间的通信模型，该模式最为适宜。

考察图 10.1，以便于我们更好地理解异步消息设计模式的流程。在该图中，服务 A 构建与服务 B、服务 C 和服务 D 之间的同步通信。反过来，服务 B、服务 C 和服务 D 识别到有一项任务需要执行，但是，该任务对于这些微服务的工作流的连续性来说不是必需的，因此随后它们会向队列发送消息。另一侧是服务 E 和服务 F，它们监听队列并使用发送的消息执行任务。该通信模型称作异步模型，其中，消息被发送且不需要任何类型的响应。这正是异步消息设计模式所使用的工作类型。

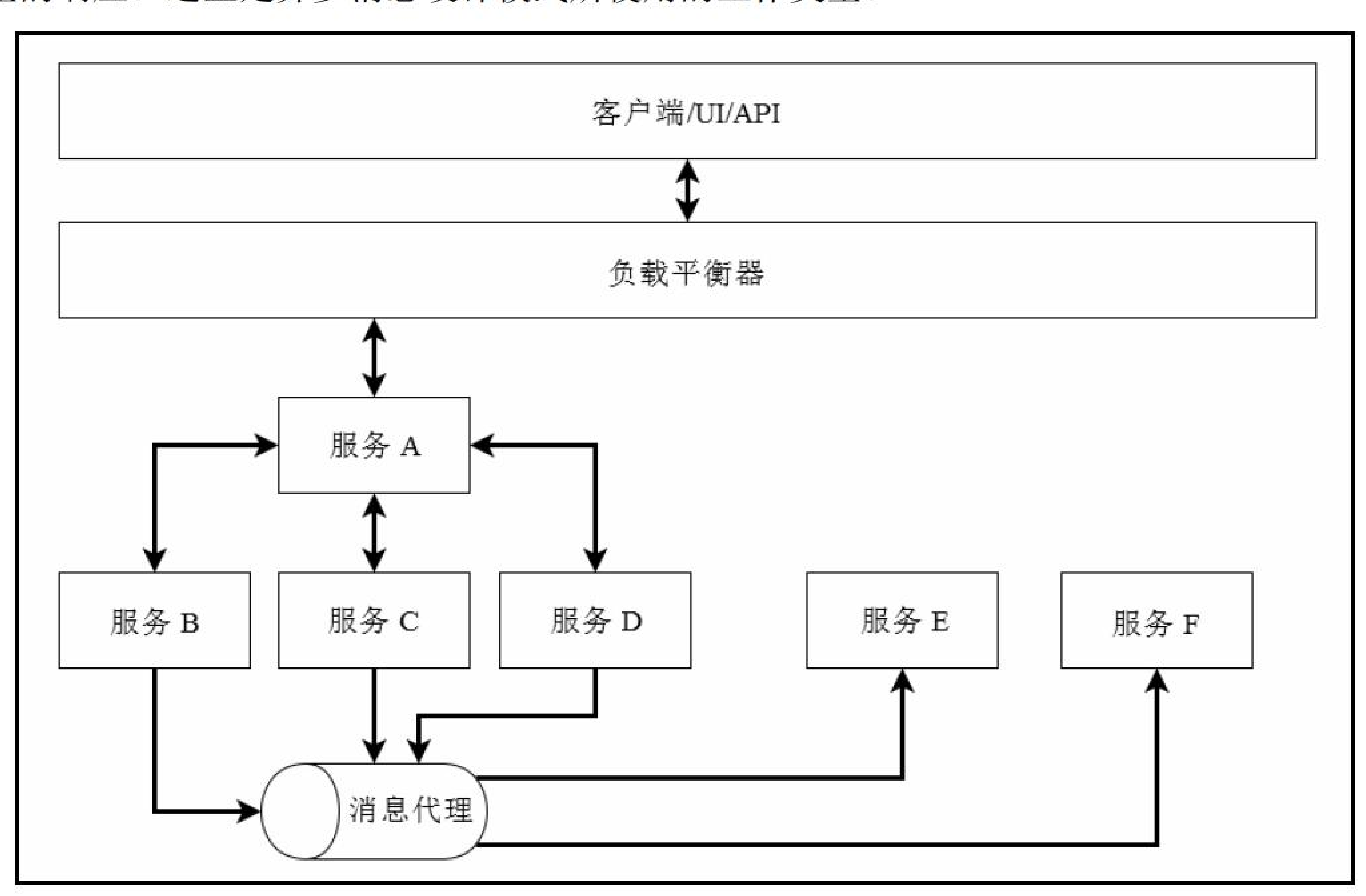


图 10.1

前述内容讨论了异步消息设计模式的行为和工作方式，下面将在应用程序中实现该模式。

当在新闻门户网站中使用异步消息设计模式时，需要创建一个新的微服务，并负责为应用程序的用户存储推荐内容。对此，微服务 RecommendationService 通过两种方式工作，即监听发送至队列中的消息，并使用直接连接到应用程序代理的 HTTP API。通过这种方式，可存储与既定用户新闻首选项相关的信息，并能够查询由应用程序 API 直接记

录的首选项。

除了 RecommendationService 微服务使用异步消息生成推荐内容之外，该微服务还将使用同步调用并通过 UsersService 微服务的数据完成推荐信息。

图 10.2 显示了微服务的分布方式。不难发现，该图与图 10.1 类似，并可用于理解异步消息设计模式的操作方式。

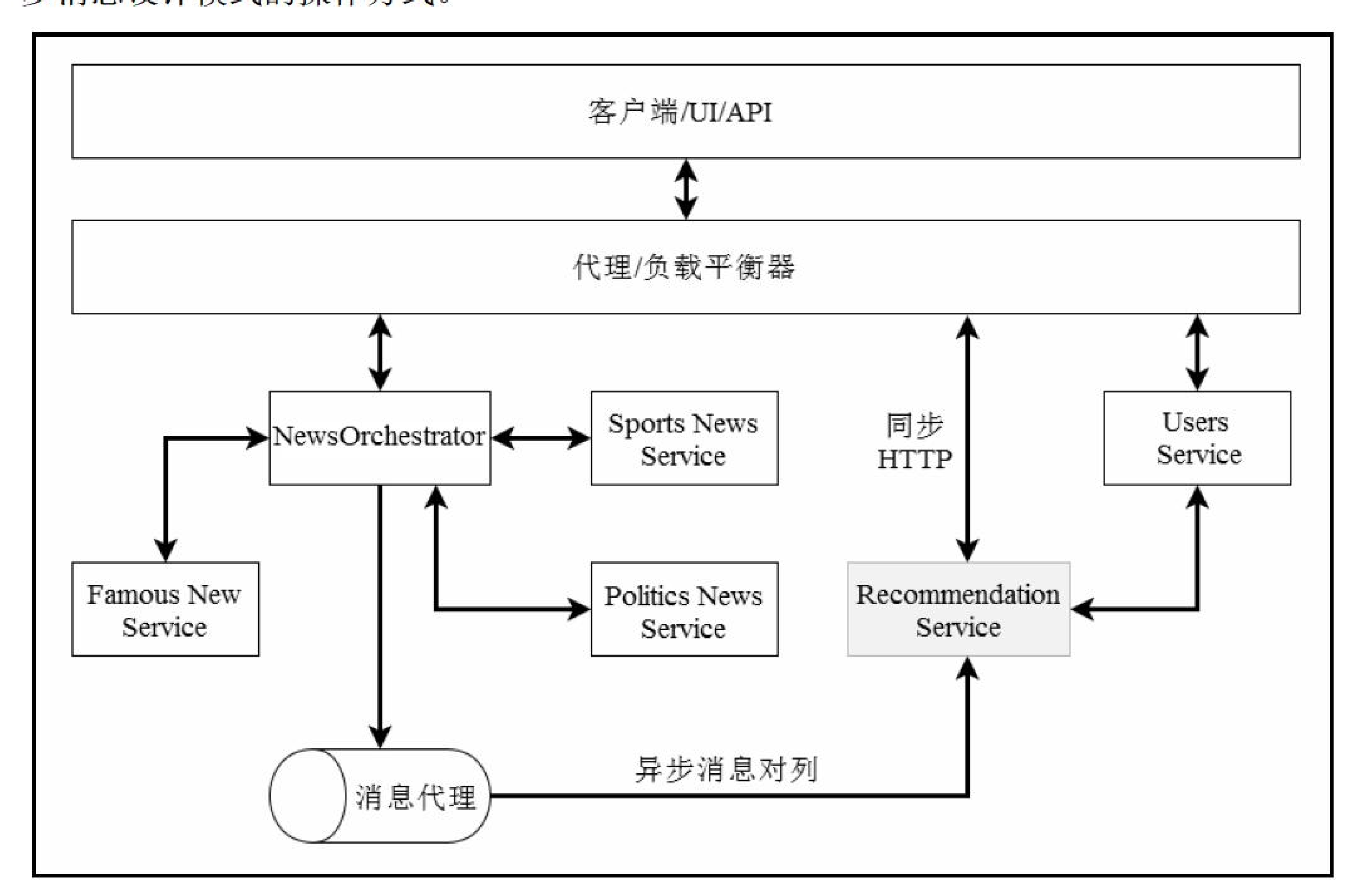


图 10.2

10.2　域定义——RecommendationService

通过 RecommendationService 微服务示例，我们已经了解到可在当前应用程序中采用异步消息设计模式。当然，这是我们所创建的一项非常简单的服务，并且与实际产品中推荐系统微服务的复杂性相去甚远。因此，RecommendationService 是一个基于演示目的的微服务，旨在实现异步消息设计模式的应用。

在了解了这一点后，下面将定义业务内容以及 RecommendationService 域。相关理念十分简单——当新闻门户网站用户搜索特定新闻内容时，该新闻中的标记将与用户 ID 相

关联。通过关联标记上的 ID，我们可以了解用户最喜欢的主题是什么，并以此推荐新闻内容和定制网页。

10.3　域定义——RecommendationService

下面将通过创建一个基于所用数据库的容器，开始着手开发 RecommendationService 微服务。在当前微服务中，将使用基于图形的数据库 Neo4j。除了易于实现之外，该数据库的行为十分有趣，因而可视为当前应用程序的完美示例。

在 docker-compose.yml 文件中，我们将稍作调整并向栈中加入 Neo4j，如下所示：

```
recommendation db:
  image: neo4j:latest
  ports:
    - "7474:7474"
    - "7687:7687"
  environment:
    NEO4J_AUTH: "none"
```

通过上述代码，Docker 容器中即包含了 Neo4j 实例。

10.4　微服务编码

当前已经设置了 RecommendationService 微服务的数据库，该数据库定义并创建于 docker-compose.yml 文件中。此处，我们将执行相同的操作并针对微服务创建一个容器，同时再次编辑 docker-compose.yml 文件。

在对应代码中，除了一些环境变量定义外，还包含了一个依赖项、数据库和消息代理，主要用于连接到数据库和队列，如下所示：

```
recommendation service:
  image: recommendation service
  build: ./RecommendationService
    volumes:
      - './RecommendationService:/app'
    environment:
      - QUEUE HOST=amqp://guest:guest@rabbitmq
      - DATABASE URL=http://recommendation db:7474/db/data
      - USER_SERVICE_ROUTE=http://172.17.0.1/user/
```

```
    depends on:
      - recommendation db
      - rabbitmq
    links:
      - recommendation db
      - rabbitmq
```

下面创建 RecommendationService 目录和文件。最终，微服务结构如下所示：

```
├── RecommendationService
│   ├── Dockerfile
│   ├──   init  .py
│   ├── config.yaml
│   ├── models.py
│   ├── requirements.txt
│   ├── service.py
```

下面将编写 RecommendationService 微服务代码，并编辑 Dockerfile。Dockerfile 代码与 News 微服务的代码相同，这是因为此处使用了相同的框架，在当前示例中为 nameko。

```
FROM python:3.6.1
COPY . /app
WORKDIR /app
RUN pip install -r requirements.txt
ENTRYPOINT ["nameko"]
CMD ["run", "--config", "config.yaml", "service"]
EXPOSE 5000
```

下一步是编写 config.yaml 文件，该文件负责通知 nameko 与其协同工作的设置类型。由于我们还将使用 nameko 并基于 HTTP 协议进行通信，因而会设置相关定义，例如 worker 的数量，以及 nameko 将在哪一个路径上响应请求。考察下列代码：

```
AMQP URI: 'amqp://guest:guest@rabbitmq'
WEB_SERVER_ADDRESS: '0.0.0.0:5000'
max_workers: 10
parent_calls_tracked: 10
LOGGING:
    version: 1
    handlers:
        console:
            class: logging.StreamHandler
    root:
        level: DEBUG
        handlers: [console]
```

前述内容设置了创建应用程序实例的文件，下面将构建 requirements.txt 文件并设置依赖关系。具体来说，该文件包含 4 个依赖关系，分别是 pytest（单元测试工具）、nameko（表示为应用程序框架）、Py2neo（表示为应用程序和 Neo4j 间的驱动程序）以及 requests（其功能之前有所介绍）。requirements.txt 文件中的相关内容如下所示：

```
pytest
nameko
py2neo
requests
```

待全部配置文件准备完毕后，接下来将编写 models.py 文件。考虑到所用数据库的特点，该文件与其他 models.py 文件稍有不同，且并不是由实体构成的模型。相反，该文件表示为一组函数，并与数据库中的数据协同工作。

第一步是在 models.py 文件中添加导入语句，此处的重点是数据库的导入操作。由于当前数据库使用了图形，因而不可将无法识别的类型导入数据库中；相应地，我们需要创建图形模型中的关系，如下列代码所示：

```
import os
from py2neo import (
    Graph,
    Node,
    Relationship,
)
```

因此，我们在代码中声明了业务结构化的常量。之前已经了解到，针对当前业务内容，对应关系为应用程序用户和新闻标记之间的关系。其中，关系类型通常表示为推荐内容，如下所示：

```
USERS_NODE = 'Users'
LABELS_NODE = 'Labels'
REL_TYPE = 'RECOMMENDATION'
```

随后，可通过 docker-compose.yml 文件中设置的环境变量创建数据库连接，如下所示：

```
graph = Graph(os.getenv('DATABASE_URL'))
```

models.py 文件中的第 1 个函数用于获取用户节点，并作为参数传递 user_id，如下所示：

```
def get_user_node(user_id):
    return graph.find_one(
        USERS_NODE,
        property_key='id',
```

```
        property value=user id,
    )
```

第 2 个函数与第 1 个函数类似，但通过 label 参数搜索节点，如下所示：

```
def get label node(label):
    return graph.find one(
        LABELS NODE,
        property key='id',
        property value=label,
    )
```

第 3 个函数负责获取包含用户关系的全部标记。针对此类搜索，我们将 user_id 用作参数。需要注意的是，在执行关系搜索之前，须针对用户节点进行搜索。利用用户节点，可通过对应标记搜索关系。考察下列代码示例：

```
def get labels by user id(user id):
    user_node = get_user_node(user_id)
    return graph.match(
        start node=user node,
        rel type=REL TYPE,
    )
```

第 4 个函数与第 3 个函数较为类似，其差别在于将搜索与某个标记相关的全部用户，如下所示：

```
def get users by label(label):
    label node = get label node(label)
    return graph.match(
        start node=label node,
        rel type=REL TYPE,
    )
```

在编写了创建查询的所有参数后，接下来将定义负责生成数据库数据的函数。

如果节点尚未在数据库中创建，第 1 个函数将在 Neo4j 中生成一个用户节点，如下所示：

```
def create user node(user):
    # get user info from UsersService
    if not get user node(user['id']):
        user node = Node(
            USERS NODE,
            id=user['id'],
            name=user['name'],
```

```
            email=user['email'],
        )
        graph.create(user_node)
```

第 2 个函数执行与第 1 个函数系统的处理操作，但会创建标记节点，如下所示：

```
def create_label_node(label):
    # get user info from UsersService
    if not get_label_node(label):
        label_node = Node(LABELS_NODE, id=label)
        graph.create(label_node)
```

第 3 个函数将创建用户/标记以及标记/用户关系。当在两侧运行关系处理时，将支持运行于两侧的搜索处理操作，否则将会出现问题。对应代码如下所示：

```
def create_recommendation(user_id, label):
    user_node = get_user_node(user_id)
    label_node = get_label_node(label)
    graph.create(Relationship(
        label_node,
        REL_TYPE,
        user_node,
    ))

    graph.create(Relationship(
        user_node,
        REL_TYPE,
        label_node,
    ))
```

下一个步骤是编写 service.py 文件代码，其工作方式类似于某种微服务控制器。

类似于微服务中的其他 Python 文件，需要声明相应的导入语句。此处最大的亮点是通过导入 **nameko** 处理程序予以实现的，这也是第一次针对 **HTTP** 处理程序使用到 **nameko**，如下所示：

```
import json
import logging
import os
import requests

from nameko.web.handlers import http
from nameko.events import event_handler
```

```
from models import (
    create user node,
    create label node,
    create recommendation,
    get labels by user id,
    get users by label,
)
```

在数据包导入语句之后，下面将编写消息代理中的消息读取器。下列代码定义了一个常规类，其中包含了 receiver 方法，以及一个装饰器，并将该方法转换为消息代理的处理程序。另外，读者也可参考代码注释以进一步理解每个处理步骤。

```
class Recommendation:

    name = 'recommendation'

    # declaring the receiver method as a handler to message broker
    @event handler('recommendation sender', 'receiver')
    def receiver(self, data):
        try:
            # getting the URL to do a sequential HTTP request to UsersService
            user service route = os.getenv('USER SERVICE ROUTE')
            # consuming data from UsersService using the requests lib
            user = requests.get(
                "{}{}".format(
                    user service route,
                    data['user_id'],
                )
            )
            # serializing the UsersService data to JSON
            user = user.json()
            # creating user node on Neo4j
            create user node(user)
            # getting all tags read
            for label in data['news']['tags']:
                # creating label node on Neo4j
                create label node(label)
                # creating the recommendation on Neo4j
                create recommendation(
                    user['id'],
                    label,
                )
```

```
        except Exception as e:
            logging.error('RELATIONSHIP_ERROR: {}'.format(e))
```

待接收器准备完毕后，下面开始编写 API 代码，并负责显示 RecommendationService 所注册的推荐内容。对应代码表示为一个类，并用作装饰器以针对 HTTP 调用生成路径，如下所示：

```
class RecommendationApi:

    name = 'recommnedation_api'
```

RecommendationApi 类中的第 1 个方法表示为 get_recommendations_by_user，该方法接收 user_id 作为参数，并返回与对应用户相关的标记，如下所示：

```
@http('GET', '/user/<int:user_id>')
    def get_recommendations_by_user(self, request, user_id):
        """Get recommendations by user_id"""
        try:
            relationship_response = get_labels_by_user_id(user_id)
            http_response = [
                rel.end_node()
                for rel in relationship_response
            ]
            return 200, json.dumps(http_response)
        except Exception as ex:
            error_response(500, ex)
```

RecommendationApi 类中的第 2 个方法表示为 get_users_recommendation_by_label，该方法将接收标记参数，并响应与该标记相关的全部用户 ID，如下所示：

```
@http('GET', '/label/<string:label>')
    def get_users_recomendations_by_label(self, request, label):
        """Get users recommendations by label"""
        try:
            relationship_response = get_users_by_label(label)
            http_response = [
                rel.end_node()
                for rel in relationship_response
            ]
            return 200, json.dumps(http_response)
        except Exception as ex:
            error_response(500, ex)
```

在文件的末尾，下列函数可帮助处理一些可能的异常。

```
def error_response(code, ex):
    response_object = {
        'status': 'fail',
        'message': str(ex),
    }
    return code, json.dumps(response_object)
```

10.5　微服务通信

RecommendationService 微服务是一种全新的微服务，具有我们已经知道的通信特性和其他新特性。该微服务将包含一个带有 HTTP 端点的 API，以及一个消息读取器，用于读取从 NewsOrchestrator 发送的消息。

发送的消息以异步方式呈现，并通过分派器传递。这样，发送消息的微服务不会得到等待响应的阻塞线程。

消息传递和非阻塞通信系统是异步消息设计模式的核心。基本上，该模式主要关注的是通信层。考虑到此模式所用的微服务具有异步特性，因而采用弹性或高效的重试机制的通信工具是非常重要的。在当前微服务中，可使用包含了事务型消息传递系统的 RabbitMQ。当然，我们也可尝试使用其他工具，但对应工具应具备应有的弹性。

为了能够更好地使用异步消息传递设计模式，除了所选用的工具之外，还应全面理解域内的业务内容，并关注异步通信内容，同时尽可能地发挥该模式的功效。

显然，异步消息传递设计模式不仅仅是一种可用的通信方式。除此之外，它还具备开发完全独立的微服务的能力。其中，响应的返回结果与业务内容无关。

10.5.1　使用消息代理和队列

与之前所处理的模式不同，在使用异步消息传递设计模式时，实际上并不存在数据编排或响应整合模型。鉴于当前模式中的异步通信模型，因而不存在响应类型，所以也不包含数据编排。

实际上，微服务的工作标准可描述为：从队列中接收消息，或者接收消息并根据结果执行相关任务。

这里，重要的是要记住使用哪种类型的工具来发送消息。如果消息具有很高的临界级别，则可针对消息处理采用事务型工具。相应地，ActiveMQ、RabbitMQ、Kafka 均为

较好的异步通信平台。

在当前应用程序中，已经采用了 RabbitMQ 并将继续对其加以使用。然而，还需对 NewsOrchestrator 代码进行适当调整，旨在收到显示新闻文章的请求时，可将用户 ID 以及用户请求的新闻数据发送至 RecommendationService 微服务中。

10.5.2 准备 pub/sub 结构

在软件架构中，发布-订阅或者 pub/sub 表示为一种消息传递模式。其中，消息发送方（称为发布者）不会为特定的接收者（称为订阅者）直接制定消息，但是会对类中发布的消息进行分类，且并不了解哪一个订阅者将接收消息。类似地，订阅者一个或多个类表示出“兴趣”，并只接收感兴趣的消息，且对发布者一无所知。

对于异步消息传递设计模式来说，发布/订阅消息模式是最适合的模式之一。

下面对 NewOrchestrator 稍作调整，并通过 pub/sub 将消息发送至 RecommendationService 微服务中。

全部调整工作将在 views.py 文件中进行，且无须使用其他新框架——nameko 用作与 News 微服务进行通信，并在 pub/sub 中用于 RecommendationService。

首先需要修改导入语句，同时导入另一个 nameko 方法，即 event_dispatcher。在文件开始处，可输入下列代码行：

```
from nameko.standalone.events import event_dispatcher
```

据此，即可使用 pub/sub 结构。就语义来讲，我们将把名称和常量从 CONFIG_RPC 更改为 BROKER_CONFIG。通过这种方式，所有的 views.py 代码都需要修改该常量的名称，如下所示：

```
BROKER_CONFIG = {'AMQP_URI': os.environ.get('QUEUE_HOST')}
```

启用 pub/sub 结构的最后一个步骤是 get_single_news 函数。在该函数中，将添加分配器。

下面仔细考察相关变化内容。从结构上讲，该函数并未产生显著变化，只是添加了分配器代码而已。

首先，我们使用从 nameko 导入的函数创建一个分派器。接下来，将把常量 BROKER_CONFIG 传递至同一函数中。相应地，nameko 方法将返回 dispatcher 对象，如下所示：

```
dispatcher = event_dispatcher(BROKER_CONFIG)
```

下面将进一步完善当前消息内容。发送至 RecommendationService 微服务的消息由搜

索新闻的用户 ID 构成。该 user_id 从请求 cookie 中获取，就好像我们正在模拟登录的用户一样。构成当前消息的其他内容还涉及包含所有新闻数据的 JSON，如下所示：

```
dispatcher('recommendation sender', 'receiver', {
    'user id': request.cookies.get('user id'),
    'news': response object['news'],
})
```

在将分配器投入使用后，get_single_news 函数的格式如下所示：

```
@news.route('/<string:news type>/<int:news id>', methods=['GET'])
def get single news(news type, news id):
    """Get single user details"""
    try:
        response object = rpc get news(news type, news id)
        dispatcher = event_dispatcher(BROKER_CONFIG)
        dispatcher('recommendation_sender', 'receiver', {
            'user_id': request.cookies.get('user_id'),
            'news': response_object['news'],
        })
        return jsonify(response object), 200
    except Exception as e:
        return erro_response(e, 500)
```

在 nameko 这样的简单框架的帮助下，通过保持应用程序的简单性，pub/sub 系统已准备就绪。

当前，无论何时调用 NewsOrchestrator 微服务的 get_single_news 方法，都将把消息发送至 RecommendationService 微服务所等待的队列。在数据经由 RecommendationService 处理后，最终结果如图 10.3 所示。

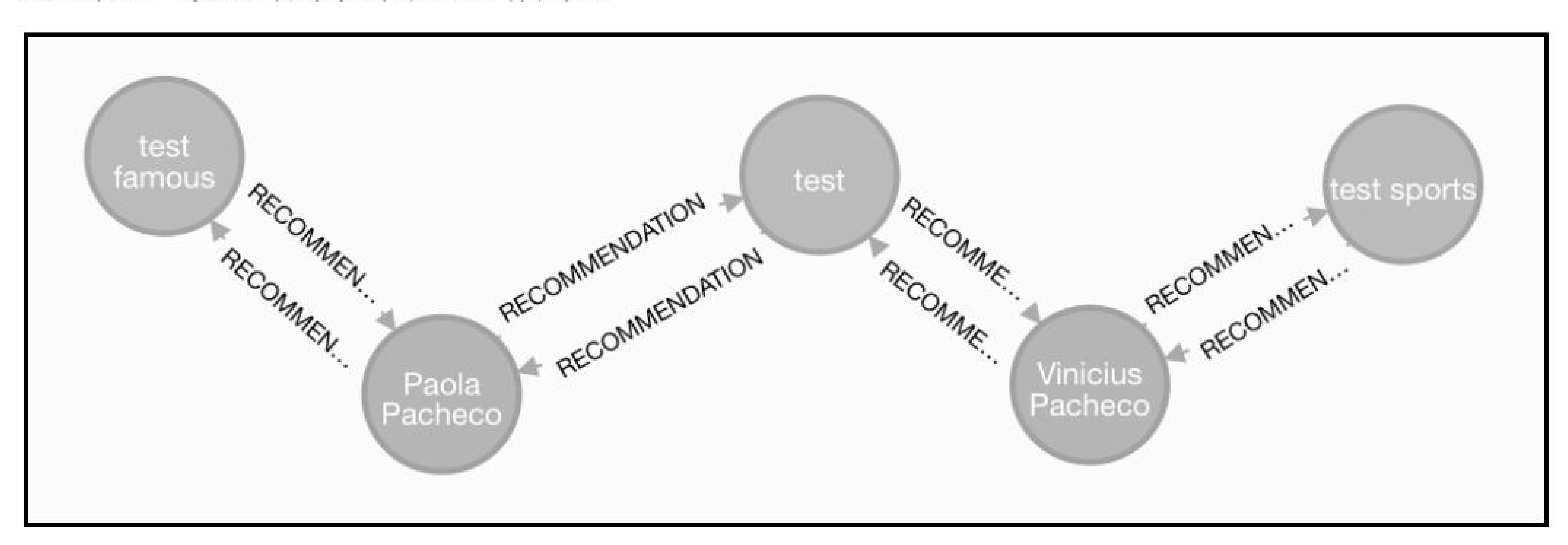


图 10.3

在图 10.3 中，可以看到两个用户执行了调用以查看两篇新闻文章。具体来说，图中包含了在 test 和 test sports 上请求的 Vinicius Pacheco，以及在 test 和 test famous 上请求的 Paola Pacheco。很快，包含标记 test 的新闻将更适合于这两名用户。

10.6　模式的可扩展性

在扩展立方体的全部轴向上，异步消息传递设计模式均提供了可扩展支持。可扩展性可以通过更多实例、更强大的机器来运行处理过程、扩展数据的分布等。然而，异步消息传递设计模式还具有其他微服务通信模式所不具备的功能，即执行延迟处理的能力。

当在微服务间使用异步通信模型时，即可在微服务中创建延迟处理。考虑到延迟处理的工作特性，与顺序工作状态下的微服务相比，延迟微服务的实例数量一般较少。

下面根据扩展立方体进一步考察异步消息传递设计模式的可扩展模型。

- 当采用异步消息传递设计模式时，*y* 轴一般是首选项。但是，对于延迟处理，或者处理过程较为繁重时，其速度均较为缓慢。
- 当 *y* 轴不能提供正确的设计模式时，*x* 轴通常会变得较为突出。在这种情况下，*y* 轴和 *x* 轴常会同时出现。
- *z* 轴也可加以使用，但其应用完全取决于微服务业务域的类型。

毫无疑问，就可扩展性而言，异步消息传递设计模式是最为简单的一种模式。

10.7　进程序列反模式

进程序列反模式可视为 OOP 生态圈中已知反模式的一个微服务版本，即顺序耦合。当某个调用依赖于另一个调用的执行时，即会产生进程序列。

对于与异步调用协同工作的微服务，此类行为有时并非是反模式。然而，对于包含异步特征的微服务来说，这在微服务通信和域中视为一种错误。

与顺序耦合不同（其中，一个类依赖于另一个类中的方法），在微服务中，进程序列依赖于微服务处理的最终结果，并使用之前在另一个微服务中处理过的数据。

我们所讨论的内容并非是一个微服务异步调用另一个微服务这一类情形——其中，两个微服务在单独的队列中对各层进行监听，但某个微服务需要知道另一个微服务结束处理后方可继续执行。

图 10.4 显示了产生于应用程序微服务层中的反模式。

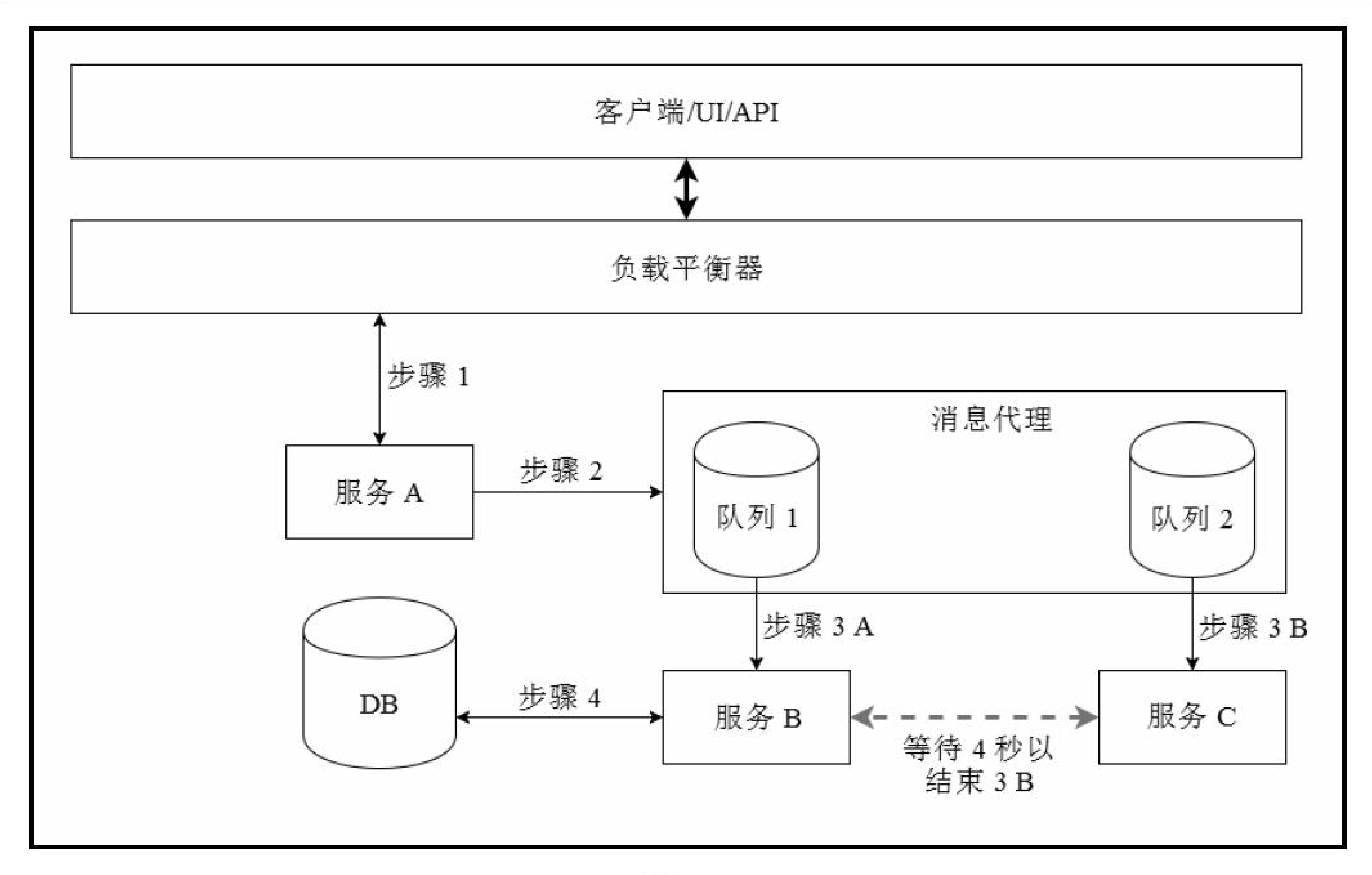


图 10.4

10.8　最佳实践方案

异步消息传递设计模式涵盖了较高级别的复杂度，但其扩展性则较好。

对于该模式，通过一些最佳实践方案，可简化理解异步消息的结构。

10.8.1　应用程序定义

对应用程序来说，应用异步消息传递设计模式是非常健康的，但是有必要将应用程序作为一个整体来考虑，以便运用该模式尽可能多地访问微服务。

通常，微视图常需要与其他微服务协同工作，这也是 DDD 如此重要的原因所在。但在某些时候，宏观视角可帮助我们理解应用程序的通信点，在哪些地方可以实现异步操作，以及在哪些地方不需要执行同步通信。

我们常希望在微服务的交互过程中提供完整的响应，但实际上，这可能并非必需。如果应用程序具有较高的成熟度和弹性，则无须等待某项任务的全部处理过程来返回 OK

响应。相应地，OK 响应可能出现于处理的开始阶段。该 OK 响应的含义从“数据被成功处理”转变为“所处理的数据已被接收”。这可视为一个重要的范式迁移，且在可扩展性方面非常有用。

10.8.2　不要尝试创建响应

针对异步调用生成响应，其维护过程异常复杂。如果对应域需要返回完整的响应，作为一个信号，其微服务不应采用异步消息。

构建包含异步消息的请求/响应模板将会破坏异步消息设计模式。

这里，创建异步消息传递中的请求/响应意味着，须针对每条接收到的消息实现一个 ID 模板，并在代理另一部分中形成一个队列，进而生成 pub/sub 双路路径。这将增加复杂度且呈现为增长势态。

10.8.3　保持简单性

异步通信中的简单性对于应用程序整体维护十分重要。类似于微服务间同步通信的死星问题，该问题同样会出现于异步通信中。但对于异步通信，快速识别故障点则要稍微复杂一些。

10.9　异步消息传递设计模式的优缺点

在开始阶段，异步消息传递设计模式理解起来较为复杂，但该模式提供了较好的扩展性。另外，深入理解该模式，将相关概念与工具相结合并付诸于实践，都将使应用程序更具扩展性以及弹性，从而降低该模式的复杂度。

该模式的优点主要包括：

- 独立的可扩展性。
- 可最大程度实现可扩展性。
- 延迟处理。
- 微服务访问的封装行为。

尽管如此，异步消息设计模式也存在一些缺点，如下所示：

- 对请求监视的复杂性较大。
- 在开始阶段，该模式理解起来较为困难。
- 难以调试。

异步消息传递设计模式的复杂性不应该成为一种障碍。良好的域设计以及适用的工具，对于采用该模式的、微服务的整体操作来说十分重要。在异步消息传递设计模式中，一些初始问题都可通过该模式提供的可扩展性得到缓解。

10.10 本章小结

本章讨论了一种高性能模式，对于 n 个应用程序，异步消息传递设计模式所带来的好处是巨大的。

在第 11 章中，将进一步完善当前项目，并在微服务间使用二进制通信。

第 11 章　微服务间的协同工作

前述内容讲述了各种概念、模式并编写了大量的代码，本章将对此予以适当整合，同时利用已经实现的所有模式对当前项目进行整体考察。

相应地，本章将对当前项目进行适当调整，进而向读者展示微服务概念的整合和演化方式，并进一步复习、巩固我们所学的知识。

本章主要涉及以下内容：

- ❑ 通用工具。
- ❑ 通信。
- ❑ 模式分布。
- ❑ 故障策略。
- ❑ API 集成。

11.1　理解当前应用程序状态

当前，新闻门户网站中的所有微服务均已编写完毕。根据每项微服务，我们可执行预期任务，并以一种可扩展的方式使其作为单一应用程序工作，且兼具弹性和良好的性能。

应用程序的使用者并不了解所使用的内容，但需要充分意识到软件连接的所有链接方式，同时还需注意应用程序中的多个关键点。

对此，我们应理解应用程序中的各部分内容，且会同时涉及软件的宏观和微观内容。为了更好地理解所开发的内容，可将应用程序划分为 3 个主要部分，如下所示：

- ❑ 公共饰面层。
- ❑ 内部层。
- ❑ 通用工具。

这将涵盖微服务架构所用方案中的所有一般概念。图 11.1 显示了新闻门户网站分布方式的高层概览图。

图中并未涉及任何模式描述，无论是通信模式抑或是微服务的内部模式。

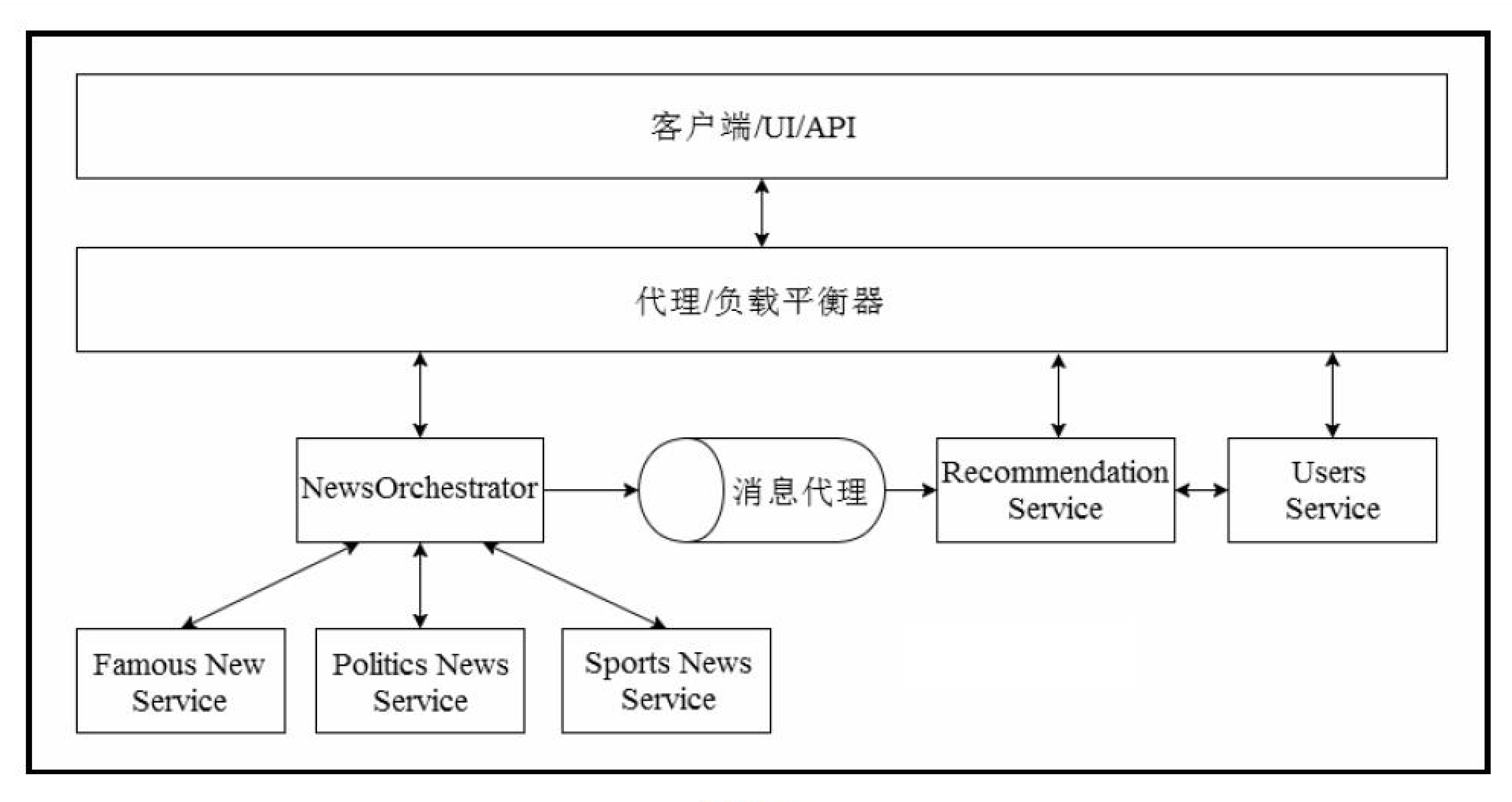


图 11.1

11.1.1　公共饰面层

公共饰面层包含了 OrchestratorService、UsersService 和 RecommendationService 3 项微服务。在一些特定时刻，最后两项微服务也用于内部层中。

需要注意的是，无论何时与微服务协同工作时，首先需要理解最外层。通过这一方式，将易于理解终端用户所期望的响应类型。永远记住，任何软件（在工程项目之前）都应是实际问题的一种解决方案。图 11.2 显示了 OrchestratorService 的内部构成方式。

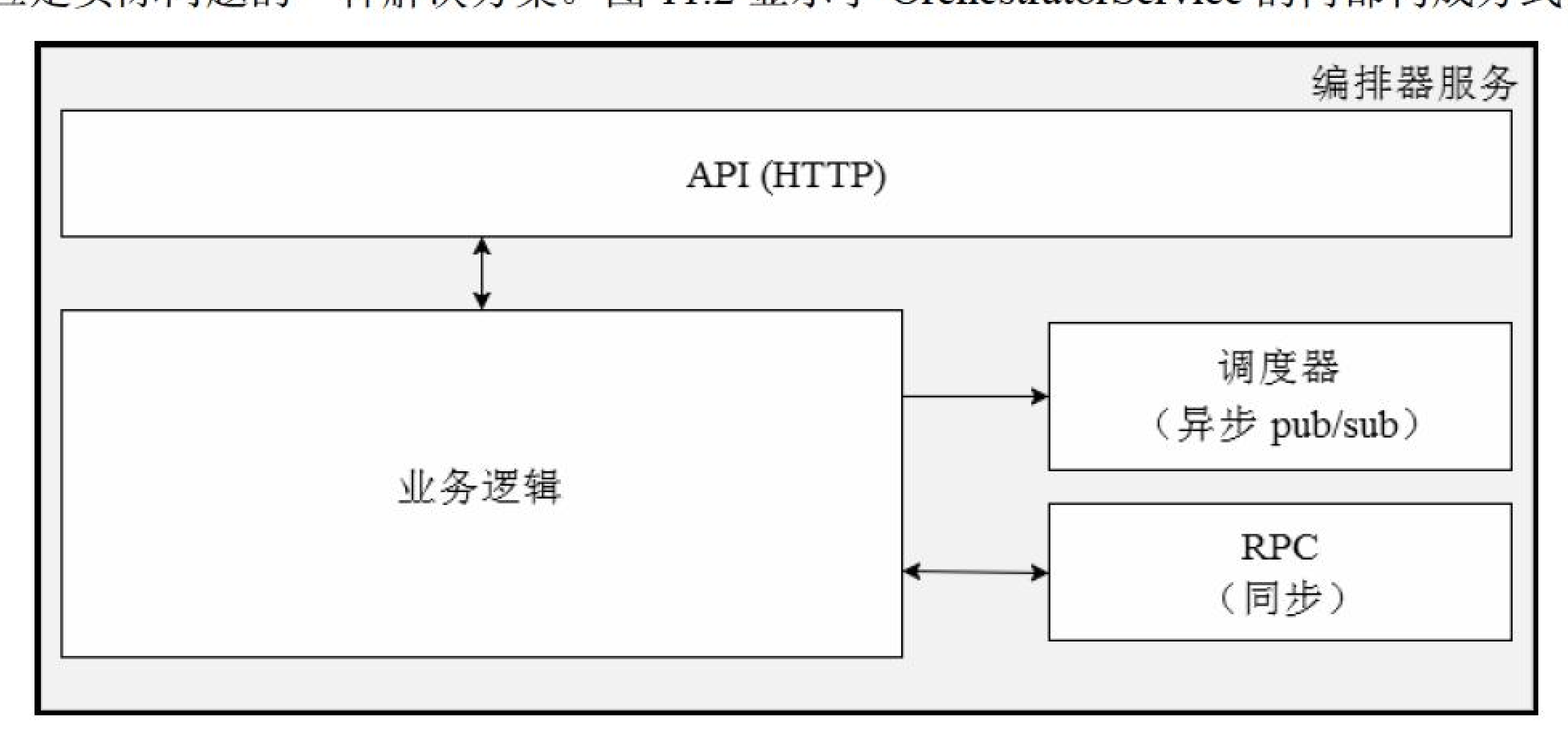


图 11.2

下面开始对 OrchestratorService 加以整体分析。对于内部层中的某些微服务来说，OrchestratorService 微服务可视为一个数据编排器，对应逻辑仅包括解释调用，并将它们重定向到负责生成信息的微服务处。对应的微服务具有用于与外部客户端通信的 HTTP API、用于微服务之间同步通信的 RPC，以及用于与其他微服务异步通信的 pub/sub 模型。另外，OrchestratorService 并不包含存储层，但能够实现缓存机制，这也使得大量的请求无须重定向至内部层。

图 11.3 显示了 UsersService 的内部构成方式。

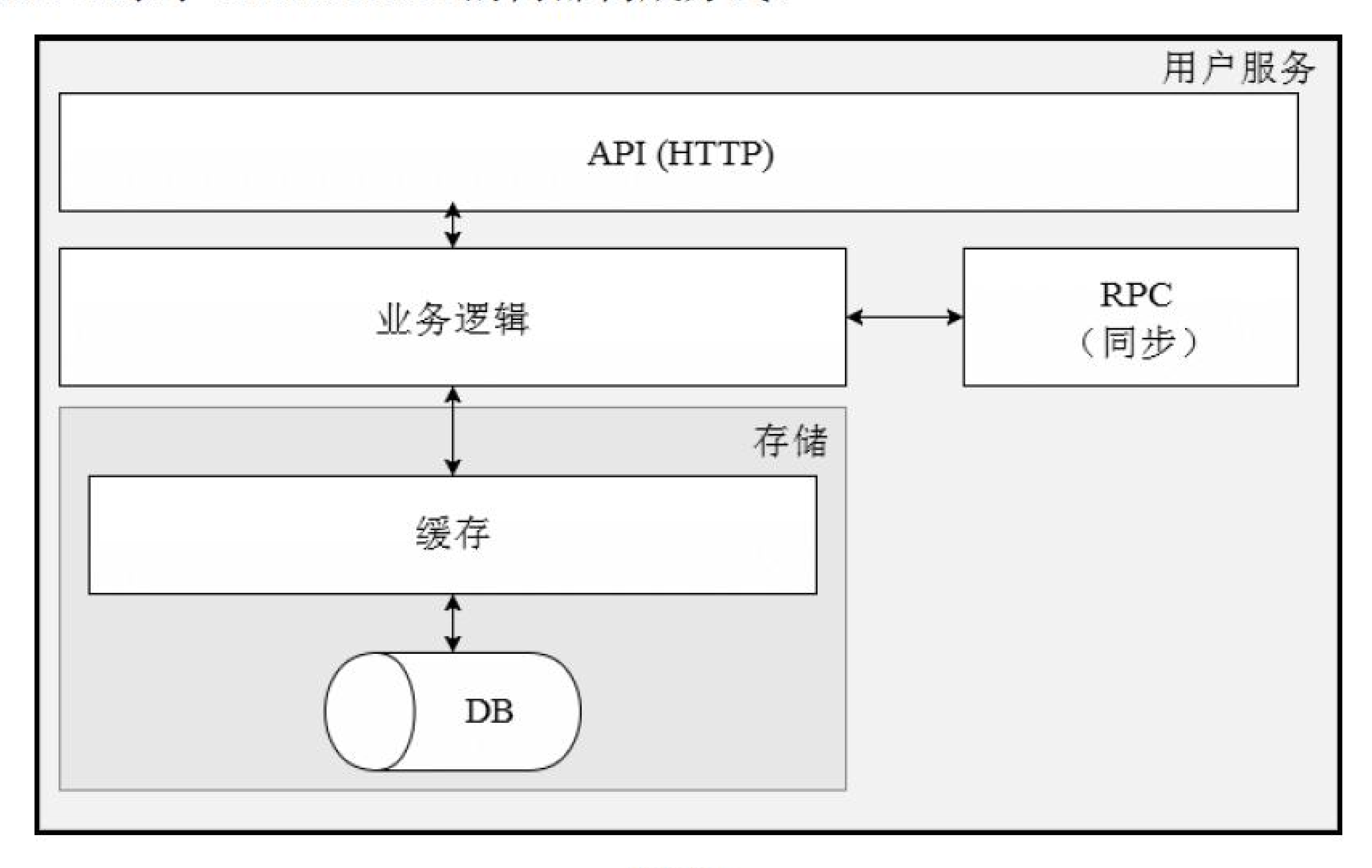


图 11.3

对于 UsersService 来说，情况则稍有不同，其中包含了持久层和缓存，同时还可使用内部模式。除此之外，该微服务包含了缓存优先这一内部模式，使得持久化数据首先存储于缓存中，并于随后存储于发送至数据库中的异步处理中。该微服务针对内部持久化采用了 Redis 作为队列工具，并通过并发模式（goroutines）执行弹性控制。UsersService 包含了两个通信层，即针对 API 层的 HTTP，以及针对内部层的 RPC 同步通信。图 11.4 显示了 RecommendationService 的内部构成方式。

RecommendationService 是一个具有独特功能的微服务，因为所有的微服务处理都可以看作是一个内部层；但是由于 API 的存在，RecommendationService 也处于公共饰面层中，因为它直接从代理接收请求。RecommendationService 微服务的通信则是通过 HTTP-API 这一方式实现的，并通过消息代理（作为信息接收者）使用 pub/sub；同时还

包含了同步 RPC 以构建与 UsersService 之间的通信。RecommendationService 微服务存在一个使用 NoSQL 数据库的简单存储层，且不存在任何内部模式。

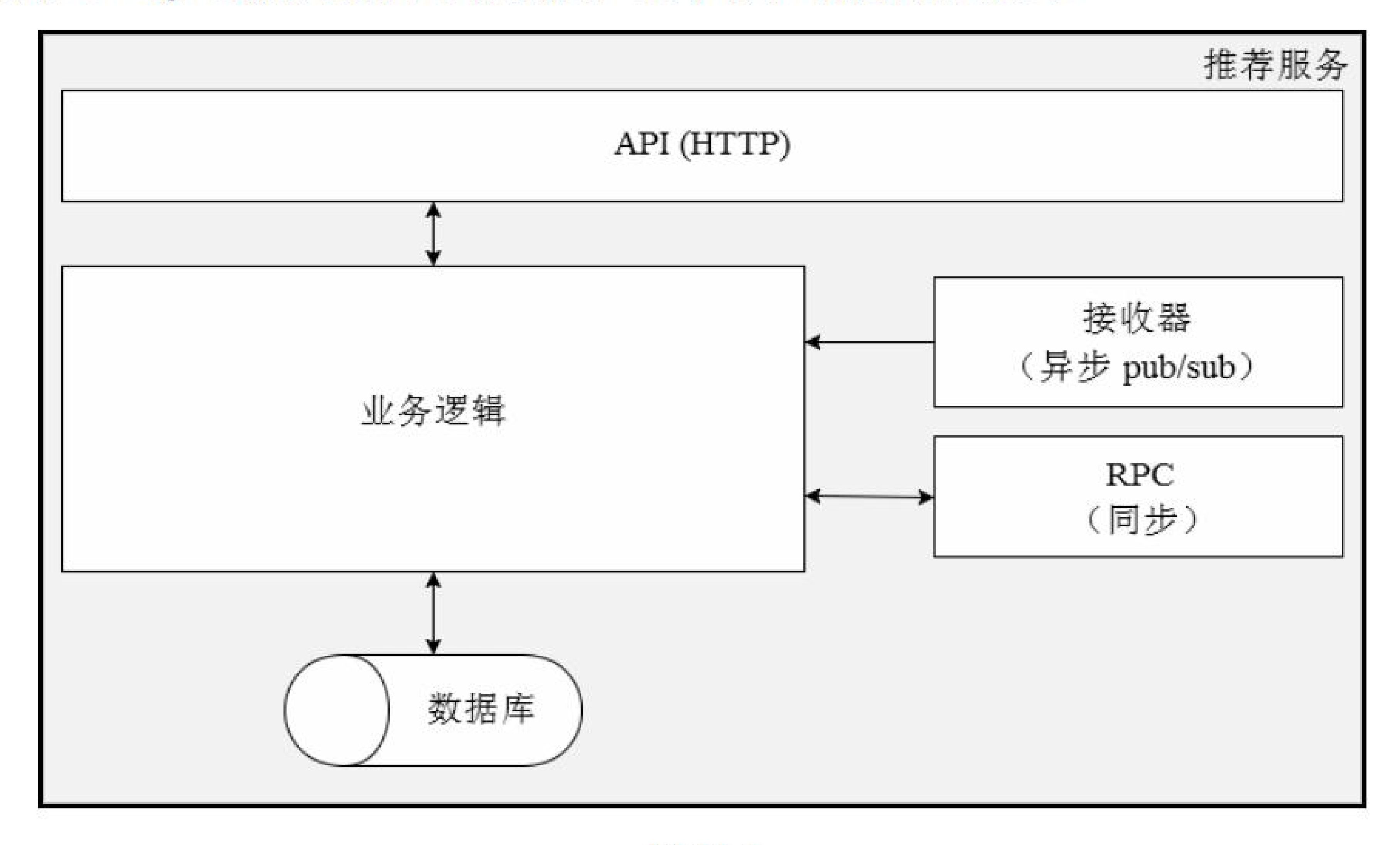


图 11.4

11.1.2　内部层

当与内部层协同工作时，存在两种类型的微服务，即 RecommendationService 以及负责处理新闻的微服务。

前述内容已经展示了 RecommendationService 的内部操作，下面将考察该内部层会话中的新闻微服务。

此处存在 3 种新闻微服务，即 FamousNewsService、PoliticsNewsService 和 SportsNewsService。虽然这 3 种微服务的主题均面向于新闻内容，但仍需要各自予以创建，以使其分别在业务和技术方向上不断演化。

然而，截至目前，此类微服务的架构仍保持一致。所有的新闻服务均设置了两个数据库，进而实现 CQRS 各事件源。对于通信层，我们采用了基于顺序同步消息传递的 RPC。

这一类微服务未包含任何类型的 API，以实现对应用程序代理的外部访问，并且都连接到 OrchestratorService 微服务以公开数据，如图 11.5 所示。

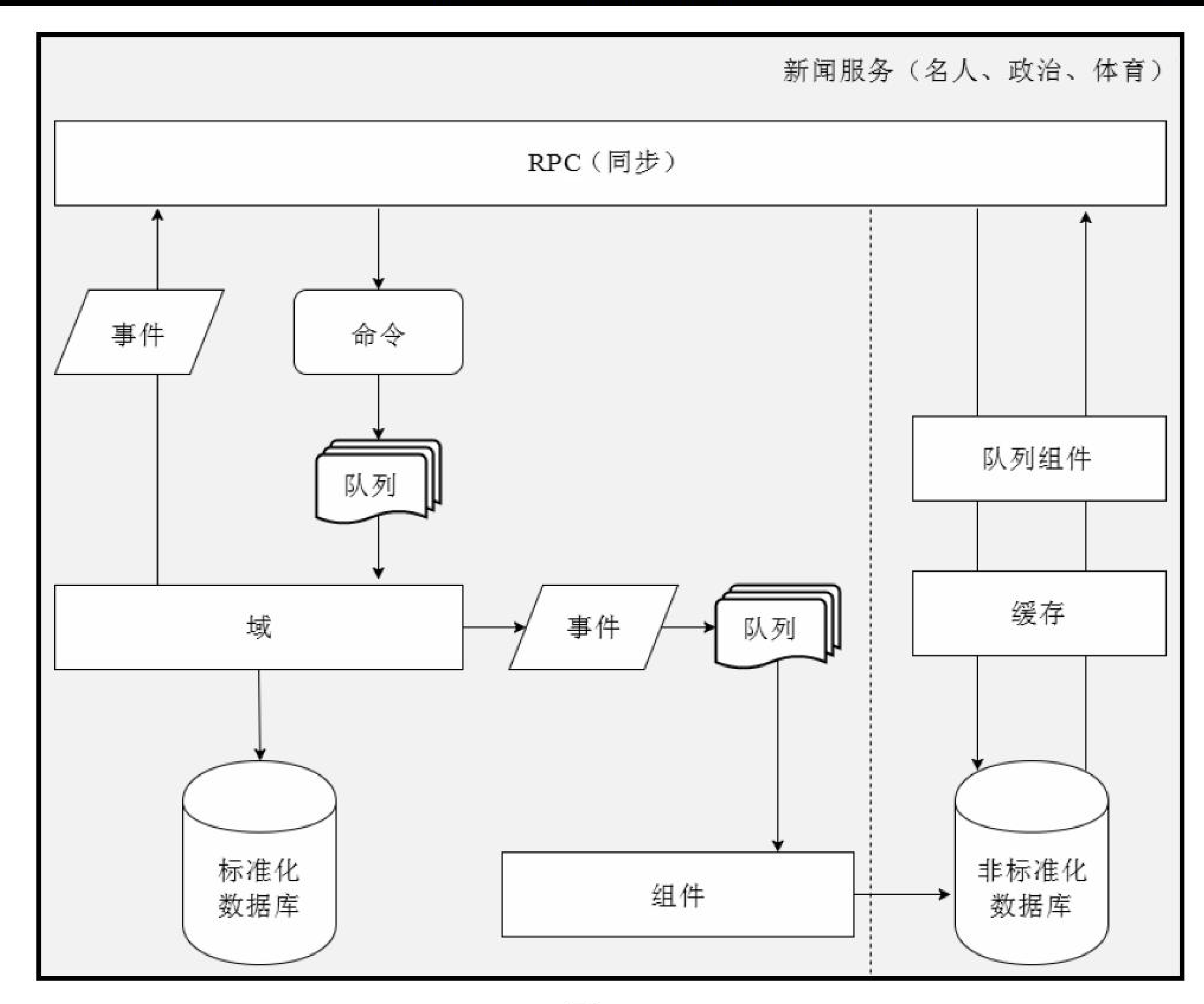


图 11.5

11.1.3　理解通用工具

这里必须要强调通用工具的用途，因为一些工具不仅可以帮助我们开发应用程序，而且还允许采用某些模式；在某些情况下，其行为类似于某种服务类型。

在我们的堆栈中，代理以及消息代理是两个主要的工具，在当前示例中，分别为 RabbitMQ 和 Nginx。

其中，消息代理可启用 OrchestratorService 和新闻微服务间的同步 RPC 通信，并支持 OrchestratorService 和 RecommendationService 间的、基于 pub/sub 模型的异步消息传递通信。

到目前为止，相同的代理工具已经为负载平衡器提供了配置。在本例中，这一工具是 Nginx。通过使用 Nginx 这一方案，可将该工具用作应用程序的实际内容，而不仅仅是

一种服务器类型。

11.2 通信层和服务间的委托

当前，微服务间的通信包含如图 11.6 所示的调用模型。

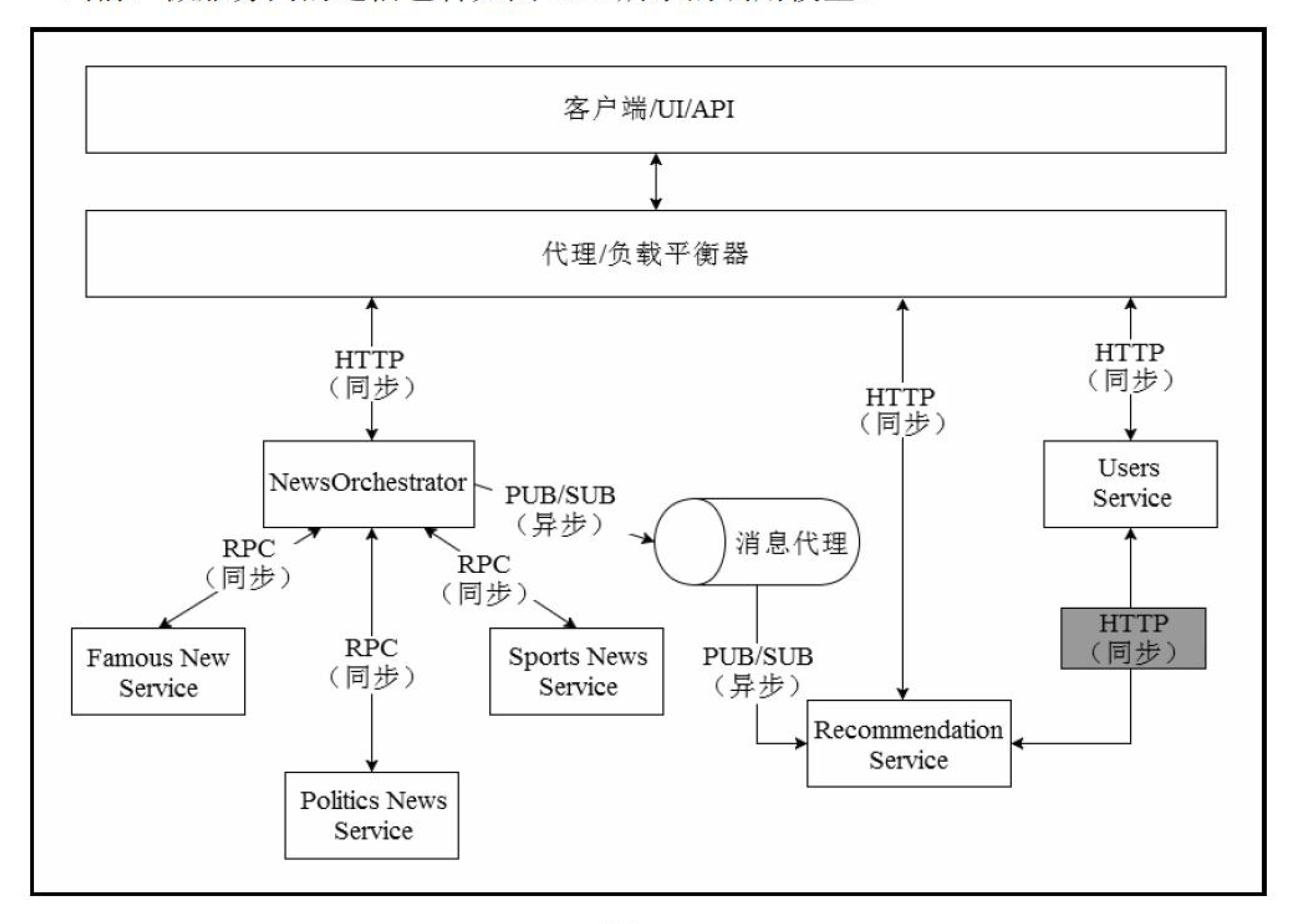


图 11.6

不难发现，针对内部层，我们采用了 RPC；而对公共饰面层产生的调用，则使用了 HTTP 调用。在内部层中的某一点处，我们采用了 HTTP 协议。该方案工作正常，但在微服务间调用的 RPC 是最为有效的，尤其是使用某种类型的数据包装器时，例如二进制协议，或者其他机制以减少数据包的数量。

下面将修改对 RPC 的调用，且有别于现有的 RPC，因为此处将传递一个二进制协议数据包。对此，我们将使用 gRPC，并分别在 UsersService 和 RecommendationService 中创建服务器和客户端。

这里需要删除的 HTTP 调用位于 service.py 文件中的、Recommendation 类中的 receiver

方法内，如下所示：

```
...
    def receiver(self, data):
        ...
            user = requests.get(
                "{}{}".format(
                    user_service_route,
                    data['user_id'],
                )
            )
        ...
...
```

该 HTTP 调用是通过 Python 库 requests 予以执行的。

11.2.1　理解服务间的数据合约

在修改不需要的 HTTP 调用之前，下面首先了解一下微服务之间的数据合约。

假设我们创建了一个应用程序，调用某个微服务完成业务任务，并阅读了与之通信和实现客户端调用的微服务文档。在尝试与微服务集成时，将会得到一条错误信息。微服务签名中的某些内容与用于创建客户机的文档并不完全相同。

上述场景十分常见。微服务一直处于变化中，而微服务签名问题也屡见不鲜。通常，这种不一致情况多出现于公司内部不同的开发团队之间。

如果文档未过期，则存在一个公共文件用于创建服务器和客户端。利用该文件，所有的服务器/客户端样板代码均不再需要。正是通过这种方式，二进制协议传输工具才得以工作。

对于微服务来说，二进制协议工具的优势与微服务间的认证相关，这意味着，采用属性、参数和方法定义的文件，需要在两个或多个微服务间的通信中加以处理，进而创建微服务间的某种合约。通过这一方式，如果微服务无法知晓或者违背了某项服务的签名，那么，问题只能在于使用了错误的验证文件。

对于 RecommendationService 和 UsersService 间的通信，下面定义微服务间的合约文件。对此，将使用到 gRPC（对应网址为 https://grpc.io）。在当前项目中，须生成一个名为 ProtoFiles 的目录，并于其中创建用于 gRPC 的文件。

在创建了 ProtoFiles 目录后，还将创建 user_data.proto 文件，该文件包含了通信所需的全部签名。

首先，在文件中写入将使用的协议缓冲版本，如下所示：

```
syntax = "proto3";
```

service 通过用于通信的方法执行当前处理。需要注意的是，GetUser 方法接收一个特定类型的参数（定义为一个请求），并作为响应发送另一个特定类型，如下所示：

```
service GetUserData {
   rpc GetUser (UserDataRequest) returns (UserDataResponse) {}
}
```

下面生成 GetUser 所需的输入类型，该类型由一个 32 位的整数构成，如下所示：

```
message UserDataRequest {
   int32 id = 1;
}
```

最后编写特定的响应类型，如下所示：

```
message UserDataResponse {
   int32 id = 1;
   string name = 2;
   string email = 3;
}
```

完整的文件内容如下所示：

```
syntax = "proto3";
service GetUserData {
   rpc GetUser (UserDataRequest) returns (UserDataResponse) {}
}

message UserDataRequest {
   int32 id = 1;
}

message UserDataResponse {
   int32 id = 1;
   string name = 2;
   string email = 3;
}
```

不难发现，上述文件短小精悍，但对于当前需求来说提供了强大的功能。通过 user_data.proto 以及安装后的 gRPC，即可执行对应的命令行，进而创建服务器和客户端。需要说明的是，该服务器采用 Go 语言编写，而客户端则利用 Python 语言加以编写。

在.proto 文件所处的同一路径下，可在终端上运行下列命令行：

```
    $ protoc --go_out=plugins=grpc:. *.proto
    $ python -m grpc_tools.protoc -I. --python_out=. --grpc_python_out=. user_data.proto
```

对于微服务间的通信，上述两行命令创建了全部所需的代码。其中，第一行代码生成 Go 文件；第二行代码则生成 Python 文件。在上述命令行运行完毕后，对应文件内容如下所示：

```
├── user_data.pb.go
├── user_data_pb2.py
└── user_data_pb2_grpc.py
```

这里，user_data.pb.go 文件须发送至 user_data 文件夹的 UsersService 目录中。相应地，UsersService 微服务对应目录结构如下所示：

```
├── Dockerfile
├── Godeps
│   ├── Godeps.json
│   └── Readme
├── Makefile
├── app.go
├── cache.go
├── db
│   ├── Dockerfile
│   ├── create.sql
│   └── dbmigrate.go
├── main.go
├── main_test.go
├── models.go
├── user_data
│   └── user_data.pb.go
```

user_data_pb2.py 和 user_data_pb2_grpc.py 文件须发送至 RecommendationService 的根目录中。RecommendationService 目录结构如下所示：

```
├── Dockerfile
├── __init__.py
├── config.yaml
├── models.py
├── requirements.txt
├── service.py
```

```
├── tests.py
├── user client.py
├── user_data_pb2.py
└── user_data_pb2_grpc.py
```

11.2.2　使用二进制通信

当使用所创建的服务器/客户端文件，并在当前项目中进行适当定位后，即可编辑 UsersService 微服务，并利用 gRPC 提供用户数据。

重构是微服务中较为常见的处理过程。对此，我们将调整 app.go 文件，进而实现缓存和数据库中用户搜索的复用功能。当前，该搜索位于 getUser 方法中，并包含两个代码块。

其中，第一个代码块直接从缓存中进行搜索，如下所示：

```
...
if value, err := a.Cache.getValue(id);err == nil && len(value) != 0 {
    log.Println("from cache")
    w.Header().Set("Content-Type", "application/json")
    w.WriteHeader(http.StatusOK)
    w.Write([]byte(value))
    return
}
...
```

如果数据未在缓存中发现，那么，第二个代码块将搜索数据库，如下所示：

```
...
user := User{ID: id}
if err := user.get(a.DB); err != nil {
    switch err {
        case sql.ErrNoRows:
        respondWithError(w, http.StatusNotFound, "User not found")
        default:
        respondWithError(w, http.StatusInternalServerError,
            err.Error())
    }
return
}
...
```

下面将上述代码块封装至两个方法中。第一个方法是 getUserFromCachemethod，对应逻辑对应于之前修改后的代码块，如下所示：

```
func (a *App) getUserFromCache(id int) (string, error) {
    if value, err := a.Cache.getValue(id); err == nil &&
          len(value) != 0 {
        return value, err
    }
    return "", errors.New("Not Found")
}
```

第二个方法是 getUserFromDB，该方法负责从数据库中获取数据，如下所示：

```
func (a *App) getUserFromDB(id int) (User, error) {
    user := User{ID: id}
    if err := user.get(a.DB); err != nil {
      switch err {
          case sql.ErrNoRows:
              return user, err
          default:
              return user, err
      }
    }
    return user, nil
}
```

下面针对 getUser 使用上述两个方法。经适当修改后，getUser 方法包含如下内容：

```
func (a *App) getUser(w http.ResponseWriter, r *http.Request) {
    vars := mux.Vars(r)
    id, err := strconv.Atoi(vars["id"])
    if err != nil {
      respondWithError(w, http.StatusBadRequest, "Invalid product ID")
      return
    }
    if value, err := a.getUserFromCache(id); err == nil {
      log.Println("from cache")
      w.Header().Set("Content-Type", "application/json")
      w.WriteHeader(http.StatusOK)
      w.Write([]byte(value))
      return
    }
    user, err := a.getUserFromDB(id)
    if err != nil {
      switch err {
        case sql.ErrNoRows:
          respondWithError(w, http.StatusNotFound, "User not found")
```

```
        default:
          respondWithError(w, http.StatusInternalServerError,
            err.Error())
      }
      return
    }
    response,   := json.Marshal(user)
    if err := a.Cache.setValue(user.ID, response); err != nil {
      respondWithError(w, http.StatusInternalServerError,

        err.Error())
      return
    }
    w.Header().Set("Content-Type", "application/json")
    w.WriteHeader(http.StatusOK)
    w.Write(response)
}
```

在代码重构完毕后，下面将对 gRPC 筹备所需内容。在 app.go 文件中，首先向 gRPC 添加所需的导入语句，如下所示：

```
...
pb "github.com/viniciusfeitosa/BookProject/UsersService/user data"
"google.golang.org/grpc"
"google.golang.org/grpc/reflection"
...
```

此处将构建一个结构，用作逻辑处理程序以提供用户数据，如下所示：

```
type userDataHandler struct {
    app *App
}
```

下列代码将形成 gRPC 所需的响应类型。

```
func (handler *userDataHandler) composeUser(user User)
    *pb.UserDataResponse {
  return &pb.UserDataResponse{
    Id: int32(user.ID),
    Email: user.Email,
    Name: user.Name,
  }
}
```

在编写响应方法的提示后，还需编写接收请求（通过 gRPC）的相关方法。此处重点

在于依赖项注入传递的上下文和请求参数，如下所示：

```
func (handler *userDataHandler) GetUser(ctx context.Context,
    request *pb.UserDataRequest) (*pb.UserDataResponse, error) {
  var user User
  var err error
  if value, err := handler.app.getUserFromCache(int(request.Id));
       err == nil {
    if err = json.Unmarshal([]byte(value), &user); err != nil {
       return nil, err
    }
    return handler.composeUser(user), nil
  }
  if user, err = handler.app.getUserFromDB(int(request.Id));
      err == nil {
   return handler.composeUser(user), nil
  }
  return nil, err
}
```

利用就绪后的请求和响应，下面开始编写 gRPC 服务器代码。首先需要声明运行服务器的方法名，如下所示：

```
func (a *App) runGRPCServer(portAddr string) {
```

接下来可定义服务器监听器，如下所示：

```
lis, err := net.Listen("tcp", portAddr)
if err != nil {
  log.Fatalf("failed to listen: %v", err)
}
```

随后创建服务器实例并将其注册至 gRPC 中。如果未出现任何错误，即可通过 gRPC 设置通信层，如下所示：

```
 s := grpc.NewServer()
 pb.RegisterGetUserDataServer(s, &userDataHandler{app: a})
 reflection.Register(s)
 if err := s.Serve(lis); err != nil {
   log.Fatalf("failed to serve: %v", err)
 }
}
```

基于服务器 gRPC 的完整方法如下所示：

```
func (a *App) runGRPCServer(portAddr string) {
  lis, err := net.Listen("tcp", portAddr)
  if err != nil {
    log.Fatalf("failed to listen: %v", err)
  }
  s := grpc.NewServer()
  pb.RegisterGetUserDataServer(s, &userDataHandler{app: a})
  reflection.Register(s)
  if err := s.Serve(lis); err != nil {
    log.Fatalf("failed to serve: %v", err)
  }
}
```

利用所构建的服务器，我们将以透明的方式激活服务器和 UsersService。在 main.go 文件中，下列代码将添加基于 gRPC 端口的另一个常量。

```
const (
    createUsersQueue = "CREATE USER"
    updateUsersQueue = "UPDATE USER"
    deleteUsersQueue = "DELETE USER"
    portAddr         = ":50051"
)
```

在 main.go 文件中，我们将加入并发模式，以同时运行两个服务器，即 API 服务器和 gRPC 服务器，如下所示：

```
...
a := App{}
a.Initialize(cache, db)
go a.runGRPCServer(portAddr)
a.initializeRoutes()
a.Run(":3000")
...
```

与服务器层相关的最后一个步骤是在 Dockerfile 中显示 gRPC 端口。对此，可简单地将该端口添加至 UsersService 的 Dockerfile 中。鉴于端口 3000 已经存在，此处将设置 50051 端口，如下所示：

```
EXPOSE 3000 50051
```

至此，服务器的设置已经完毕，下面着手设置客户端，该客户端位于 RecommendationService 中。第一步需要创建 user_client.py 文件。在 user_client.py 文件中，首先应编写导入语句，如下所示：

```
import logging
import os
import grpc

import user data pb2
import user_data_pb2_grpc
```

下面将定义一个与 Python 中的 Context Manager 相似的类。读者可阅读下列代码中的注释内容，以了解所运行的内容。

```
class UserClient:

  def   init  (self, user id):
    self.user id = int(user id)
    # Open a communication channel with UsersService
    self.channel =
      grpc.insecure channel(os.getenv('USER SERVICE HOST'))
    # Creating stub to get data
    self.stub =
      user data pb2 grpc.GetUserDataStub(self.channel)

  def   enter  (self):
    # Call common method between both microservices passing
      the request type
    return self.stub.GetUser(
        user data pb2.UserDataRequest(id=self.user id)
    )

  def   exit  (self, type, value, traceback):
    # Logging the process
    logging.info('Received info using gRPC', [type, value, traceback])
```

待客户端准备完毕后，即可修改业务逻辑，移除 HTTP 并将 gRPC 用作通信方式。这里，逻辑变化须在 RecommendationService 的 service.py 文件中执行。

首先需要移除请求的导入操作，并使用刚刚创建的客户端导入操作，如下所示：

```
from user_client import UserClient
```

在 receiver 方法中，将利用 user_client 替换当前请求。需要注意的是，此处不再将所接收的数据转换为 JSON 格式——gRPC 的响应结果表示为.proto 文件中声明的特定类型。对应代码如下所示：

```
@event_handler('recommendation_sender', 'receiver')
def receiver(self, data):
  try:
    # consuming data from UsersService using the requests lib
    with UserClient(data['user_id']) as response:
        user = response
    # creating user node on Neo4j
    create_user_node(user)
...
```

对于 gRPC 返回的对象类型，当前无须绑定 JSON，但仍需要修改 models.py 文件中的 create_user_node 函数，以接收这一新的类型。注意，此处无须在字典中查找数据，可直接使用对象的属性（类似于类中的操作），如下所示：

```
def create_user_node(user):
  # get user info from UsersService
  if not get_user_node(user.id):
    user_node = Node(
        USERS_NODE,
        id=user.id,
        name=user.name,
        email=user.email,
  )
  graph.create(user_node)
```

最后一步是移除 docker-compose.yml 中的环境变量，以使 RecommendationService 知晓 UsersService 的路径，如下所示：

```
recommendation_service:
  image: recommendation_service
  build: ./RecommendationService
  volumes:
    - './RecommendationService:/app'
  environment:
    - QUEUE_HOST=amqp://guest:guest@rabbitmq
    - DATABASE_URL=http://recommendation_db:7474/db/data
    - USER_SERVICE_HOST=usersservice:50051
  depends_on:
    - recommendation_db
    - rabbitmq
    - usersservice
  links:
    - recommendation_db
```

```
  - rabbitmq
  - usersservice
```

在经过上述修改后，gRPC 即可实现正常工作，同时消除了内部层中的 HTTP 调用。图 11.7 显示了微服务间的通信结构。

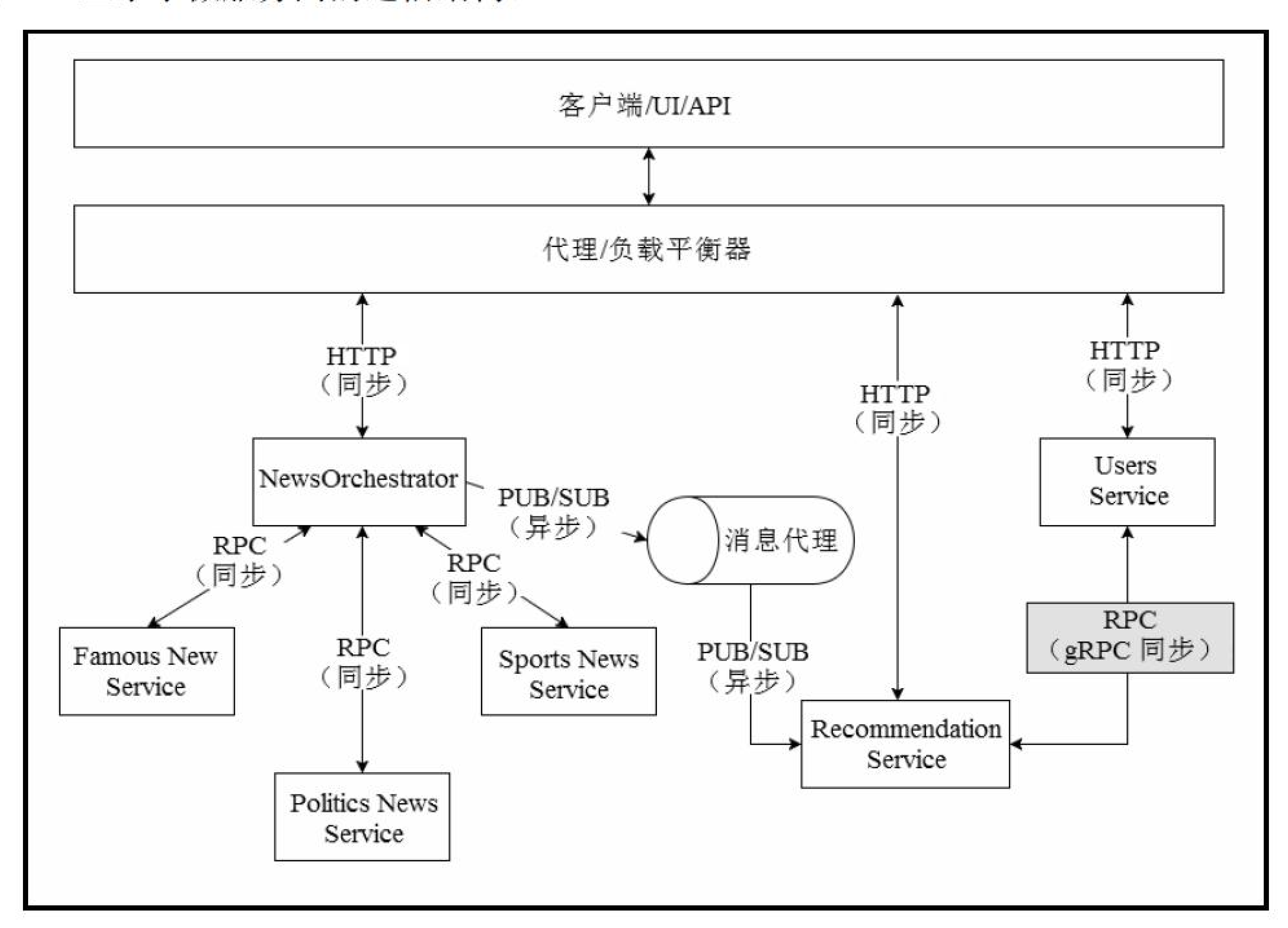


图 11.7

11.3　模式分布

本书介绍了微服务间通信的各种模式，并使用到了大部分所提及的模式。在开发过程中，我们还可以重新组织代码，同时在应用程序中对现有模式进行调整，或者增加新的模式。

在当前应用程序中，我们使用了下列模式：

- 代理微服务设计模式。该模式使用了 Nginx 以实现代理角色，同时针对 OrchestratorNewsService、UsersService 和 RecommendationService API 引用了代理。
- 聚合器微服务设计模式。OrchestratorNewsService 针对 FamousNewsService、

SportsNewsService 和 PoliticsNewsServicemicroservices 饰演了聚合器这一角色。

- 分支微服务设计模式。该模式用于构建 UsersService 和 RecommendationService 间的通信——RecommendationService 需要以异步方式使用到源自 UsersService 的信息，进而完成所处理的任务。
- 异步消息微服务设计模式。该模式用于 OrchestratorNewsService 和 RecommendationService 之间。

图 11.8 显示了一些较为重要的模式结构。

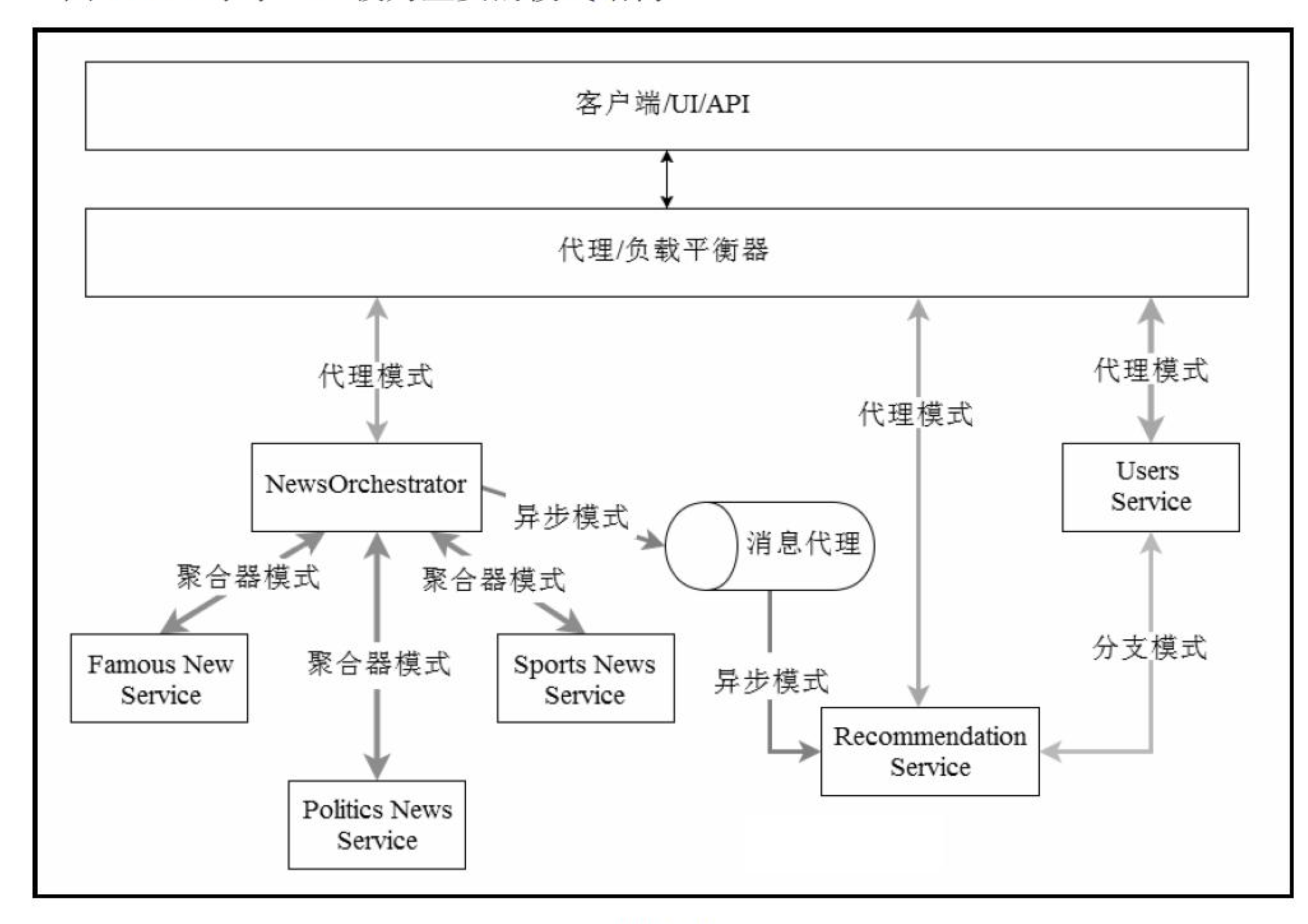


图 11.8

11.4 故 障 策 略

前述内容在应用程序开发过程中介绍了一些反模式。在反模式带来的附带损害中，对于使用微服务架构的应用程序来说，最为严重的则是微服务间的循环调用。

循环调用会阻止每个微服务的持续集成、独立部署和渐进演化，并在测试中产生一种“成功”的错觉。除了产生与应用程序其他部分同时部署这一需求之外，微服务还会

对域造成损害。

适当地使用模式可以防止在微服务策略中出现此类故障。

显然，当处理微服务间的通信时，可采用断路器或超时等技术（参见第 4 章）。

11.5　API 集成

对于微服务须执行的业务来讲，微服务 API 可视为其网关。如果微服务 API 存在任何问题，业务也会随之受到影响。

通常情况下，API 的问题主要源于：微服务签名中的变化没有得到正确的通知。为了避免此类问题，第一种方案是在服务数据的应用程序和使用数据的应用程序之间设置一个公共文件。在当前微服务中，gRPC 可满足这一要求。

另一种可成功地集成微服务 API 的方法是采用 API 版本控制。下面再次考察 nginx.conf 文件。其中，位置设置处于不一致状态且不包含版本指示信息，如下所示：

```
...
 server {
   listen 80;

   location / {
   ...
   }

   location /news/ {
   ...
   }

   location /recommendation/ {
   ...
   }
...
```

下面对此进行修改，并设置我们的 API 版本。当前，对于每个上游 Nginx 指令，均拥有一个版本定义的位置，如下所示：

```
...
upstream users servers {
  server bookproject usersservice 1:3000;
}
upstream orcherstrator_servers {
```

```
  server bookproject orcherstrator news service 1:5000;
}
upstream recommendation servers {
  server bookproject recommendation service 1:5000;
}
server {
  listen 80;

  location /users/v1/ {
    proxy pass http://users servers/;
    ...
  }

  location /news/v1/ {
    proxy pass http://orcherstrator servers/;
    ...
  }

  location /recommendation/v1/ {
    proxy pass http://recommendation servers/;
    ...
  }
}
```

假设需要将微服务 UsersService 修改为一个新的版本，我们将暂时保留两条路径，直到所有 API 使用者完全迁移到新版本的 API 中，如下所示：

```
upstream users servers {
  server bookproject usersservice  1:3000;
}

upstream users_servers_v2 {
  server bookproject_usersservice_v2_1:3000;
}

server {
  listen 80;

  location /users/v1/ {
    proxy pass          http://users servers/;
    ...
  }
```

```
  location /users/v2/ {
    proxy_pass          http://users_servers_v2/;
    ...
  }
}
```

11.6 本章小结

本章介绍了微服务的当前状态，并在应用程序中创建了二进制模型和版本控制端点。

第 12 章将讨论可应用于微服务上的某些测试类型，其中包括集成测试、单元测试和其他一些测试，它们不仅能够帮助我们发现潜在的错误，还可使应用程序更易于维护。

第 12 章　微服务测试

前述章节在新闻门户网站应用程序中实现了多种模式，本章将对项目业务流程是否可正常工作进行验证。

即使对于一个不是非常大的微服务项目，执行手动测试也会异常复杂，且肯定会在评估过程中失败，或者是遗忘应用程序业务中的某些重要任务。

毫无疑问，自动化测试适用于任何类型的软件。在本章中，将讨论以下主题：

- 单元测试。
- 集成测试。
- 端到端测试。
- 管线。
- 签名测试。
- Monkey 测试。

12.1　单 元 测 试

单元测试早已被人们所熟知，且在软件开发行业中被视为一种规范。对于单元测试来说，其中包含了多项标准和实践方案。相应地，最佳方案则是确保运行所构建的每个软件。

单元测试可用于证明计算机程序中最小的可测试部分。从这个意义上讲，最大的挑战是编写可测试的代码；否则，将无法应用单元测试。

这里需要了解一个重要的特性，单元测试只能证明代码单元段。假设我们要测试一个与数据库通信的函数，当构建单元测试时，须采用一种机制，以使该函数无法触碰到数据库，该机制称作 mock 测试。当对某个单元测试使用 mock 测试时，须隔离需要测试的函数；随后，数据库中的任意变化均不会在单元测试中产生冲突。

一些单元测试涵盖了某些因素，并可改变其最终结果，例如数据库以及浮点时间变量，这将破坏单元测试的某些重要规则，如确定性和幂等性（idempotent）。其中，确定性单元测试可描述为：无论执行多少次，最终结果终将保持一致。相比较而言，非确定性单元测试将视为一个重大错误。当采用单元测试时，较好的实践方案是不仅要证明成

功情形，还应进一步证明潜在的缺陷。当处理异常时，这对于发现潜在的代码错误十分有用。

在我们的新闻门户网站中，考虑到问题的复杂性，将使用 OrchestratorNewsService 微服务作为单元测试应用示例。

OrchestratorNewsService 微服务中包含了公共 API 层。涵盖内部服务的 RPC 特性，并在消息代理中发布数据。所有这些特征使得测试过程更加复杂。

如前所述，基于单元测试中的相关概念，需要对测试的代码段予以隔离。需要注意的是，当前目标是进行包含成功和失败两种情况的确定性测试。下面开始针对 OrchestratorNewsService 微服务编写单元测试代码，其中将涉及更为复杂的场景，特别是表达与其他微服务间的通信时。

在 OrchestratorNewsService 目录中，须创建 tests.py 文件，随后声明所需的导入语句以创建单元测试。注意，除了导入 unittest 依赖关系之外，还应从 mock 包中导入补丁装饰器（patch decorator），这对于创建具有确定性的单元测试来说十分有用。对应代码如下所示：

```
import json
import unittest

from mock import patch

from app import app
from views import error response
from flask_testing import TestCase
```

下面定义 BaseTestCase 类，该类负责载入基本的测试设置。

```
class BaseTestCase(TestCase):

   def create app(self):
      app.config.from object('config.TestingConfig')
      return app
```

接下来创建测试，并验证开发环境设置，如下所示：

```
class TestDevelopmentConfig(TestCase):

   def create app(self):
      app.config.from object('config.DevelopmentConfig')
      return app
```

```
    def test_app_is_development(self):
        self.assertTrue(app.config['DEBUG'] is True)
```

随后是验证测试配置的测试内容，如下所示：

```
class TestTestingConfig(TestCase):

    def create_app(self):
        app.config.from_object('config.TestingConfig')
        return app

    def test_app_is_testing(self):
        self.assertTrue(app.config['DEBUG'])
        self.assertTrue(app.config['TESTING'])
```

为了完成配置测试，下列代码显示了产品配置的验证过程。

```
class TestProductionConfig(TestCase)

    def create_app(self):
        app.config.from_object('config.ProductionConfig')
        return app

    def test_app_is_production(self):
        self.assertFalse(app.config['DEBUG'])
        self.assertFalse(app.config['TESTING'])
```

前述内容讨论了一些较为简单、常见的单元测试。下面考察一种较为有趣的情况，并针对新闻文章搜索创建单元测试。首先需要定义测试类，如下所示：

```
class TestGetSingleNews(BaseTestCase):
```

接下来，我们将定义相关方法，并针对搜索成功情形进行验证。需要注意的是，此处使用了补丁装饰器，从而可针对 rpc_get_news 函数和 event_dispatcher 函数创建 mock。相应地，补丁实例则通过注入依赖予以传递，如下所示：

```
@patch('views.rpc_get_news')
@patch('nameko.standalone.events.event_dispatcher')
    def test_success(self, event_dispatcher_mock, rpc_get_news_mock):
```

此处，我们需要声明由 mock 所返回的数值。其中，第一个 mock 表示为 event_dispatcher_mock，其中将传递一个匿名函数，该函数接收一些数值，但不执行任何操作。对应代码如下所示：

```
event_dispatcher_mock.return_value = lambda v1, v2, v3: None
rpc_get_news_mock.return_value = {
    "news": [
        {
            " id": 1,
            "author": "unittest",
            "content": "Just a service test",
            "created at": {
                "$date": 1514741833010
            },
            "news type": "famous",
            "tags": [
                "Test",
                "unit test"
            ],
            "title": "My Test",
            "version": 1
        }
    ],
    "status": "success"
}
```

接下来将作为参数发送一个新闻文章 ID，并以此调用 get_single_news 端点。随后，将对来自 mock 的调用予以解释，并对响应结果进行验证。对应代码如下所示：

```
With self.client:
    response = self.client.get('/famous/1')
    data = json.loads(response.data.decode())
    self.assertEqual(response.status code, 200)
    self.assertIn('success', data['status'])
    self.assertTrue(len(data['news']) > 0)
    for d in data['news']:
        self.assertEqual(d['title'], 'My Test')
        self.assertEqual(d['content'], 'Just a service test')
        self.assertEqual(d['author'], 'unittest')
```

同时，还将使用类似的处理过程验证同一 get_single_news 端点的故障点。除此之外，还将使用到补丁装饰器并向 mock 传递相关数值，但此处会强制产生一个错误——对应函数并不知晓如何使用 None 值进行工作。在处理过程结尾处，还将验证该错误消息是否为我们所期望的结果，对应代码如下所示：

```
@patch('views.rpc get news')
@patch('nameko.standalone.events.event_dispatcher')
```

```
def test_fail(self, event_dispatcher_mock, rpc_get_news_mock):
    event_dispatcher_mock.return_value = lambda v1, v2, v3: None
    rpc_get_news_mock.return_value = None
    response = self.client.get('/famous/1')
    data = json.loads(response.data.decode())
    self.assertEqual(response.status_code, 500)
    self.assertEqual('fail', data['status'])
    self.assertEqual("'NoneType' object is not subscriptable",
data['message'])
```

在对新闻文章的搜索过程进行测试后，还将进一步测试创建过程。对于单元测试，将再次声明一个类，如下所示：

```
class TestAddNews(BaseTestCase):
```

当前，对应补丁源自 rpc_command 函数。同样，补丁实例作为测试方法的参数并通过依赖注入予以传递，如下所示：

```
@patch('views.rpc_command')
def test_sucess(self, rpc_command_mock):
    """Test to insert a News."""
```

下面将定义一个 dict 并将其作为数据的输入，同时也是响应中的部分内容，如下所示：

```
dict_obj = dict(
    title='My Test',
    content='Just a service test',
    author='unittest',
    tags=['Test', 'unit test'],
)
rpc_command_mock.return_value = {
    'status': 'success',
    'news': dict_obj,
}
with self.client:
    response = self.client.post(
        '/famous',
        data=json.dumps(dict_obj),
        content_type='application/json',
    )
    data = json.loads(response.data.decode())
    self.assertEqual(response.status_code, 201)
    self.assertEqual('success', data['status'])
    self.assertEqual('My Test', data['news']['title'])
```

类似于验证成功情形，此处还将对错误场景予以验证。其中，None 值将被赋予至 dict_obj 中。这将产生一个错误结果，表明 add_news 所接收的负载无效，如下所示：

```
def test fail by invalid input(self):
    dict_obj = None
    with self.client:
        response = self.client.post(
            '/famous',
            data=json.dumps(dict obj),
            content type='application/json',
        )
        data = json.loads(response.data.decode())
        self.assertEqual(response.status code, 400)
        self.assertEqual('fail', data['status'])
        self.assertEqual('Invalid payload', data['message'])
```

对于单元测试，考虑各种场景是非常重要的。之前曾编写了一个测试，并验证传递错误负载时所发生的情况。但是，如果问题位于微服务中，并注册数据库中的负载，情况又当如何？下面将对该测试加以讨论。

首先，我们使用包含补丁的装饰器，并于随后创建包含有效负载的 dict，如下所示：

```
@patch('views.rpc command')
def test fail to register(self, rpc command mock):
    """Test to insert a News."""
    dict obj = dict(
        title='My Test',
        content='Just a service test',
        author='unittest',
        tags=['Test', 'unit test'],
    )
```

接下来，将创建带有某种副作用的 mock 并引发异常。我们通过调用 add_news 函数的端点来完成信息的发送过程，如下所示：

```
rpc_command_mock.side_effect = Exception('Forced test fail')
with self.client:
    response = self.client.post(
        '/famous',
        data=json.dumps(dict obj),
        content type='application/json',
    )
    data = json.loads(response.data.decode())
```

最后，将验证是否收到了来自 mock 的、上述异常所发送的消息，如下所示：

```
self.assertEqual(response.status code, 500)
self.assertEqual('fail', data['status'])
self.assertEqual('Forced test fail', data['message'])
```

TestGetAllNewsPerType 类通过单元测试对 get_all_news_per_type 函数进行验证，具体过程并无太大变化。首先是类声明，随后针对补丁使用一个装饰器、创建 mock、调用端点并验证所返回的内容。全部过程如下所示：

```
class TestGetAllNewsPerType(BaseTestCase):

    @patch('views.rpc get all news')
    def test sucess(self, rpc get all news mock):
        """Test to get all News paginated."""
        rpc get all news mock.return value = {
            "news": [
                {
                    " id": 1,
                    "author": "unittest",
                    "content": "Just a service test 1",

                    "created at": {
                        "$date": 1514741833010
                    },
                    "news type": "famous",
                    "tags": [
                        "Test",
                        "unit test"
                    ],
                    "title": "My Test 1",
                    "version": 1
                },
                {
                    " id": 2,
                    "author": "unittest",
                    "content": "Just a service test 2",
                    "created at": {
                        "$date": 1514741833010
                    },
                    "news type": "famous",
                    "tags": [
                        "Test",
```

```
                        "unit test"
                    ],
                    "title": "My Test 2",
                    "version": 1
                },
            ],
            "status": "success"
        }
        with self.client:
            response = self.client.get('/famous/1/10')
            data = json.loads(response.data.decode())
            self.assertEqual(response.status code, 200)
            self.assertIn('success', data['status'])
            self.assertEqual(2, len(data['news']))
            counter = 1
            for d in data['news']:
                self.assertEqual(
                    d['title'],
                    'My Test {}'.format(counter)
                )
                self.assertEqual(
                    d['content'],
                    'Just a service test {}'.format(counter)
                )
                self.assertEqual(
                    d['author'],
                    'unittest'
                )
                counter += 1
```

通过相同的方式，还可对故障场合构建单元测试，对应处理过程也基本相同，如下所示：

```
@patch('views.rpc get all news')
def test fail(self, rpc get all news mock):
    """Test to get all News paginated."""
    rpc get all news mock.side effect = Exception('Forced test fail')
    with self.client:
        response = self.client.get('/famous/1/10')
        data = json.loads(response.data.decode())
        self.assertEqual(response.status code, 500)
        self.assertEqual('fail', data['status'])
        self.assertEqual('Forced test fail', data['message'])
```

由于测试场景变得越发复杂，因而可针对简单场景定义相关类，例如针对辅助函数的测试。下列单元测试较为简单，并验证函数（负责封装错误消息的逻辑内容）是否可正常工作。

```
class TestUtilsFunctions(BaseTestCase):
    def test error message(self):
        response = error response('test message error', 500)
        data = json.loads(response[0].data.decode())
        self.assertEqual(response[1], 500)
        self.assertEqual('fail', data['status'])
        self.assertEqual('test message error', data['message'])
```

在文件的结尾处设置了一个条件，进而可执行本地 Python 测试工具，如下所示：

```
if   name   == '  main  ':
    unittest.main()
```

不难发现，上述单元测试执行起来较为简单，其复杂之处在于：针对有效的测试覆盖范围，须考虑到所有的可能场景。

为了进一步查看 OrchestratorNewsService 单元测试的运行状态，可使用下列命令行：

```
$ docker-compose -f docker-compose.yml up --build -d

$ docker exec -it $(shell docker ps -q --filter
"name=orcherstrator_news_service_1") python tests.py
```

12.2　针对集成测试配置容器

在真正运行集成测试之前，首先需要对其配置容器。可在当前开发环境中使用 docker-compose。目前，所有的应用程序设置均位于同一 docker-compose.yml 文件中。

docker-compose 可通过传递 docker-compose.yml 文件进而重写设置，如下所示：

```
$ docker-compose -f docker-compose.yml -f docker-compose.test.yml up
--build -d
```

下面将分离上述设置并创建不同的 docker-compose 文件。其中，每个文件对应于特定的环境。对此，将创建一个新的文件 docker-compose.test.yml，该文件仅包含需要重写的设置内容，如下所示：

```
version: '3'
services:
```

```
  users service:
    environment:
      -DATABASE URL=postgresql://postgres:postgres@users service db:5432/
users test?sslmode=disable

  famous news service:
    environment:
      - QUERYBD HOST=mongodb://querydb famous:27017/news test
      -COMMANDDB HOST=postgresql://postgres:postgres@commanddb famous:
5432/news test?sslmode=disable

  politics news service:
    environment:
      - QUERYBD HOST=mongodb://querydb politics:27017/news test
      -COMMANDDB HOST=postgresql://postgres:postgres@commanddb politics:
5432/news test?sslmode=disable

  sports_news_service:
    environment:
      - QUERYBD HOST=mongodb://querydb sports:27017/news test
      -COMMANDDB HOST=postgresql://postgres:postgres@commanddb sports:
5432/news test?sslmode=disable

  recommendation service:
    environment:
      - DATABASE_URL=http://recommendation_db:7474/db/test_data
```

在上述代码段中可以看到，全部修改内容仅包含了数据库路径，这是针对当前测试创建的一种特定的数据库，即沙箱。

正如创建包含特定设置项的新文件那样，我们也应移除 docker-compose.yml 文件中不必要的设置内容。

某些微服务所使用的环境变量需要进行替换，例如 main.go 文件中的 UsersServices。设置内容的替换操作如下所示：

```
...
connectionString := os.Getenv("DATABASE DEV URL") // replace this
connectionString := os.Getenv("DATABASE_URL") // by this
...
```

同时，可针对所有微服务重复该处理过程。

12.3　集 成 测 试

在测试的构造过程中，集成测试与单元测试十分类似，但前者涵盖了不同的概念。

类似于单元测试，集成测试也需要具备确定性，但不仅仅是证明隔离的代码段。在微服务环境下，集成测试将验证测试开始点至最终交互间的全部流程。例如，它可以是一个供应商应用程序或数据库。

在 OrchestratorNewsService 微服务示例中（该微服务将用作测试示例），当测试端点时，将不再创建任何一种 mock，且该过程尽可能地接近真实状态。然而，我们应保证全部测试具有确定性。对此，可使用特定的数据库，并编写良好的集成测试代码。

首先在 OrchestratorNewsService 存储库中创建 tests_integration.py 文件，编写导入语句并声明测试基类，如下所示：

```
import json
import unittest
from app import app
from flask testing import TestCase

class BaseTestCase(TestCase):

    def create app(self):
        app.config.from object('config.TestingConfig')
        return app
```

随后定义集成测试类，如下所示：

```
class TestIntegration(BaseTestCase):
```

当实例化当前类时，setUp 方法将被执行，且该方法在所有其他方法之前被执行。需要注意的是，当前针对微服务执行 HTTP POST 操作，且不包含任何 mock。这意味着，可高效地实现数据库信息的持久化操作。当运行 setUp 方法时，将保存响应结果以及实例变量所返回的 JSON。对应代码如下所示：

```
def setUp(self):
    dict obj = dict(
        title='My Test',
        content='Just a service test',
        author='unittest',
        tags=['Test', 'unit test'],
    )
```

```
    with self.client:
        self.response_post = self.client.post(
            '/famous',
            data=json.dumps(dict obj),
            content type='application/json',
        )
        self.data_post = json.loads(
            self.response post.data.decode()
        )
```

第一个集成测试仅验证 setUp 运行的 POST 信息是否正确，如下所示：

```
def test add news(self):
    """Test to insert a News."""
    self.assertEqual(self.response post.status code, 201)
    self.assertEqual('success', self.data post['status'])
    self.assertEqual('My Test', self.data_post['news']['title'])
```

第二个集成测试将调用 get_single_news，并与微服务 FamousNewsService 集成。下列代码中显示了所创建的新闻文章的 ID。

```
def test get single news(self):
    response = self.client.get(
        'famous/{id}'.format(id=self.data_post['news']['id'])
    )
    data = json.loads(response.data.decode())
    self.assertEqual(response.status code, 200)
    self.assertIn('success', data['status'])
    self.assertTrue(len(data['news']) > 0)
    self.assertEqual(data['news']['title'], 'My Test')
    self.assertEqual(data['news']['content'], 'Just a service test')
    self.assertEqual(data['news']['author'], 'unittest')
```

最后，再次设置执行集成测试的相关条件，如下所示：

```
if  name  == ' main ':
    unittest.main()
```

当需要查看 OrchestratorNewsService 集成测试的运行状态时，可使用下列命令行：

```
$ docker-compose -f docker-compose.yml -f docker-compose.test.yml up
--build -d

$ docker exec -it $(shell docker ps -q --filter
"name=orcherstrator_news_service_1") python tests_integration.py
```

不难发现，如果弃用了 mock 工具，集成测试和单元测试包含了相同的测试套件工具。另外，二者的测试过程也较类似，仅是对应的概念有所不同。

12.4　端到端测试

从概念上讲，端到端测试与集成测试较为类似，但前者将沿着应用程序的全部流程。该类型测试的主要目的在于检测各个流程阶段是否受损。关于端到端测试以及集成测试，许多开发人员仍对此产生混淆。

二者间的最大差别在于，集成测试验证应用程序的某一部分与其他微服务、工具或供应商间的集成状态；而端到端测试则验证应用程序的业务流，而非与后续内容间的集成。

另外，还有可能包含多个端到端测试，并验证不同的流程。在当前应用程序示例中，将要测试的流程包括：

（1）创建用户。

（2）针对每个新闻服务类型创建新闻文章（如名人、政策以及体育）。

（3）通过发送请求 cookie 中的 user_id 函数，搜索测试中创建的全部新闻文章。

（4）验证针对用户创建的推荐内容。

对于代码内容，首先需要在项目的根目录中创建新目录 TestRobot，并在该目录中创建 main.go 文件，该文件即为端到端测试工具。

在 TestRobot/main.go 文件中，须分别声明数据包、导入语句、常量以及测试所用的相关结构。对应代码如下所示：

```
package main

import (
	"bytes"
	"encoding/json"
	"errors"
	"fmt"
	"io/ioutil"
	"log"
	"net/http"
	"os"
	"strings"
	"time"
)
```

```
const (
	baseURL             string = "http://localhost/"
	newsURL             string = baseURL + "v1/news/"
	usersURL            string = baseURL + "v1/users/"
	recommendationURL   string = baseURL + "v1/recommendation/"
)

type News struct {
	ID       int         `json:"id"`
	Author   string      `json:"author"`
	Content  string      `json:"content"`
	NewsType string      `json:"news type"`
	Tags     []string    `json:"tags"`
	Title    string      `json:"title"`
	Version  int         `json:"version"`
}

type RespNewsBody struct {
	News     News        `json:"news"`
	Status   string      `json:"status"`
}

type Users struct {
	ID       int         `json:"id"`
	Name     string      `json:"name"`
	Email    string      `json:"email"`
	Password string      `json:"password"`
}

type RecommendationByUser struct {
	ID string `json:"id"`
}
```

下一步是编写辅助函数，以避免代码重复。其中，**newsUnmarshaler** 函数将接收的 JSON 转换为 **struct** 实例 **RespNewsBody**，如下所示：

```
func newsUnmarshaler(resp *http.Response) (RespNewsBody, error) {
	defer resp.Body.Close()
	body,   := ioutil.ReadAll(resp.Body)
	var respBody RespNewsBody
	if err := json.Unmarshal(body, &respBody); err != nil {
		return respBody, err
```

```
	}
	return respBody, nil
}
```

newsIntegrityValidator 函数负责验证所接收的数据是否为期望数据，对应代码如下所示：

```
func newsIntegrityValidator(respBody RespNewsBody, newsType string) error {
	if respBody.News.Version < 1 {
		return errors.New("News wasn't created")
	}
	if strings.Title(newsType)+" end-to-end Test" !=
respBody.News.Author {
		return fmt.Errorf("Inconsistent value checking the author:
%s", respBody.News.Author)
	}
	return nil
}
```

recommendationUnmarshaler 将所接收的 JSON 转换为 RecommendationByUser 结构实例列表，对应代码如下所示：

```
func recommendationUnmarshaler(resp *http.Response)
([]RecommendationByUser, error) {
	defer resp.Body.Close()
	body,   := ioutil.ReadAll(resp.Body)
	var respBody []RecommendationByUser
	if err := json.Unmarshal(body, &respBody); err != nil {
		return respBody, err
	}
	return respBody, nil
}
```

recommendationIntegrityValidator 函数负责验证所接收的数据是否为期望数据，如下所示：

```
func recommendationIntegrityValidator(recommendations
[]RecommendationByUser) error {
	if len(recommendations) < 3 {
		return fmt.Errorf("Fail. The quantity of recommendations was
less then spected. expected: 3 received: %d", len(recommendations))
	}
	return nil
}
```

下面开始编写一个函数，并验证在会话开始阶段制定的流程。首先须声明函数名，即 StartToEndTestMinimalFlow。对应代码如下所示：

```
func StartToEndTestMinimalFlow() {
       log.Println("### Starting minimal validation flow ###")
       log.Println("Validating user creation")
```

前述内容曾生成负载、创建用户并执行了 POST 方法，通过验证成功，则继续执行该处理过程。对应代码如下所示：

```
reqBody := []byte(`{
       "name": "end-to-end User",
       "email": "end-to-end@test.com",
       "password": "123456"
}`)
resp, err := http.Post(usersURL, "Application/json",
bytes.NewBuffer(reqBody))
if err != nil {
       os.Exit(1)
}
defer resp.Body.Close()
body,   := ioutil.ReadAll(resp.Body)
var respUser Users
if err := json.Unmarshal(body, &respUser); err != nil {
       log.Fatalln(err)
}
if respUser.Email != "end-to-end@test.com" {
       log.Fatalln("Inconsistent value checking the user email: ",
respUser.Email)
}
log.Println("User creation validated with success")
```

接下来针对新闻文章生成负载，如下所示：

```
mapNews := map[string][]byte{
       "famous": []byte(`{
              "author": "Famous end-to-end Test",
              "content": "This content is just a test using the
famous end-to-end test robot",
              "tags": ["Famous test"],
              "title": "FamousNews test end-to-end"
       }`),
       "politics": []byte(`{
```

```
                "author": "Politics end-to-end Test",
                "content": "This content is just a test using the
politics end-to-end test robot",
                "tags": ["Politics test"],
                "title": "PoliticsNews test end-to-end"
        }`),
        "sports": []byte(`{
                "author": "Sports end-to-end Test",
                "content": "This content is just a test using the
sports end-to-end test robot",
                "tags": ["Sports test"],
                "title": "SportsNews test end-to-end"
        }`),
}
newsTypeID := make(map[string]int)
```

随后，针对每个负载项执行循环操作并重复上述流程。对应代码如下所示：

```
for newsType, reqBody := range mapNews {
        log.Println("Validating news creation:", newsType)
```

编排器将使用 HTTP POST，如下所示：

```
resp, err := http.Post(newsURL+newsType, "Application/json",
bytes.NewBuffer(reqBody))
if err != nil {
        os.Exit(1)
}
respBody, err := newsUnmarshaler(resp)
if err != nil {
        log.Fatalln(err)
}
```

针对每篇新闻文章，所返回的数据内容将被验证。如果验证器未返回错误，则处理过程继续进行，如下所示：

```
if err := newsIntegrityValidator(respBody, newsType); err != nil {
        log.Fatalln(err)
}
log.Println("News creation validated with success:",newsType)
```

下面将对所创建的新闻文章执行搜索操作，并传递请求 cookie 中的 user_id，如下所示：

```
log.Println("Validating news get:", newsType)
client := &http.Client{}
```

```
req, err := http.NewRequest("GET", fmt.Sprintf("%s%s/%d",
newsURL, newsType, respBody.News.ID), nil)
if err != nil {
        log.Fatalln(err)
}
req.Header.Set("Cookie", fmt.Sprintf("user_id=%d", respUser.ID))
resp, err = client.Do(req)
if err != nil {
        log.Fatalln(err)
}
respBody, err = newsUnmarshaler(resp)
if err != nil {
        log.Fatalln(err)
}
```

再次，将验证新闻文章的完整性，并借助于下列搜索结果：

```
if err:= newsIntegrityValidator(respBody, newsType); err! = nil {
        log.Fatalln(err)
}
log.Println("Got news with success:", newsType)
```

在循环结尾处，将生成新闻类型和新闻文章 ID 间的键/值对，如下所示：

```
        newsTypeID[newsType] = respBody.News.ID
}
```

下面将针对某个用户搜索并验证新闻标记的推荐内容，如下所示：

```
    log.Println("Validating recommendations")
    time.Sleep(1 * time.Second)
    resp, err = http.Get(fmt.Sprintf("%s%s/%d", recommendationURL,
"user", respUser.ID))
    if err != nil {
            log.Fatalln(err)
    }
    recommendationByUser, err := recommendationUnmarshaler(resp)
    if err != nil {
            log.Fatalln(err)
    }
    if err := recommendationIntegrityValidator(recommendationByUser);
err != nil {
            log.Fatalln(err)
    }
    log.Println("Recommendations validated with success")
```

```
    log.Println("### Finished minimal validation flow ###")
}
```

文件结尾则定义了执行当前任务的 main 函数，如下所示：

```
func main() {
        StartToEndTestMininalFlow()
}
```

当查看端到端测试的运行状态时，可使用下列命令行：

```
$ docker-compose -f docker-compose.yml -f docker-compose.test.yml up
--build -d

$ go run ${PWD}/TestRobot/main.go
```

12.5 发布管线

这里，管线可视作处理流程，且需要在代码发布至产品，或者特定的版本控制库之前运行。

对此，较为常见的管线构建工具包括 Jenkins 以及其他持续集成（CI）工具。图 12.1 显示了较为常见的代码测试管线流程。



图 12.1

管线概念可视为一种较好的测试流控制器。

12.6 签名测试

假设下列场景：多个开发团队参与了同一应用程序中不同微服务的开发，此类微服务于其间包含了某些通信机制。对此，存在一类合约并表示为微服务的负载，有时也称作服务签名。相应地，某个团队修改了该微服务的签名将导致应用程序的其他部分出现错误。

如前所述，微服务的修改行为十分常见，尤其是微服务表示为内部层中的部分内容时。当某个微服务的签名发生变化，须针对其他团队发布一项任务——相关微服务应与

当前签名予以整合。

除了签名发生变化之外，相关问题还包括信息的缺失，其原因在于，微服务签名的整合操作可能失效或者是被遗忘。

签名测试适用于微服务有效负载中可能出现的警告内容的变化。对此，有很多方法可以做到这一点，例如 webhook、版本控制库，甚至是脚本和 CI。

作为示例，下面考察 RecommendationService 和 UsersService 间的通信。对于创建两个微服务间的 RPC 客户端/服务器通信，其间存在一个公共文件，该文件位于 ProtoFiles 库中，并称作 user_data.proto。在当前示例中，验证过程十分简单，仅需检查该文件是否被修改即可。

当验证 user_data.proto 文件的变化时，可使用下列命令：

```
$ git diff --name-only HEAD HEAD~1 | grep user_data.proto
```

上述命令验证当前提交和前一次提交的文件是否存在差异。

12.7　Monkey 测试

Monkey 测试通常是一类包含随机值的自动化测试，并关注应用程序中识别出的错误。一般情况下，这种测试中产生的错误往往是由于输入处理失败，或者（应用程序压力所导致的）输入处理缓慢而造成的。

Monkey 测试常与一种称之为 Fuzzing 的测试技术结合使用。Fuzzing 依赖于向计算机程序输入无效、意外和随机的数据，并于随后监控程序，分析运行期错误这一类异常。

总体而言，Monkey 对于识别一些现有的错误十分有效。

12.8　Chaos Monkey

Chaos Monkey（对应网址为 https://github.com/Netflix/chaosmonkey）由 Netflix 工程团队发布，顾名思义，该测试以一种随机方式产生“混乱”。其处理过程将随机选取生成环境中的服务器，并在工作期内对其予以禁用，并以此度量应用程序的弹性。

当采用 Chaos Monkey 时，可以确定如何更好地分发服务器，寻找更高效的监控系统，并开发弹性模式。

假设某个应用程序实现了 CQRS。如果 Chaos Monkey 测试从写入数据库中关闭了服



务器，情况又当如何？如图 12.2 所示。

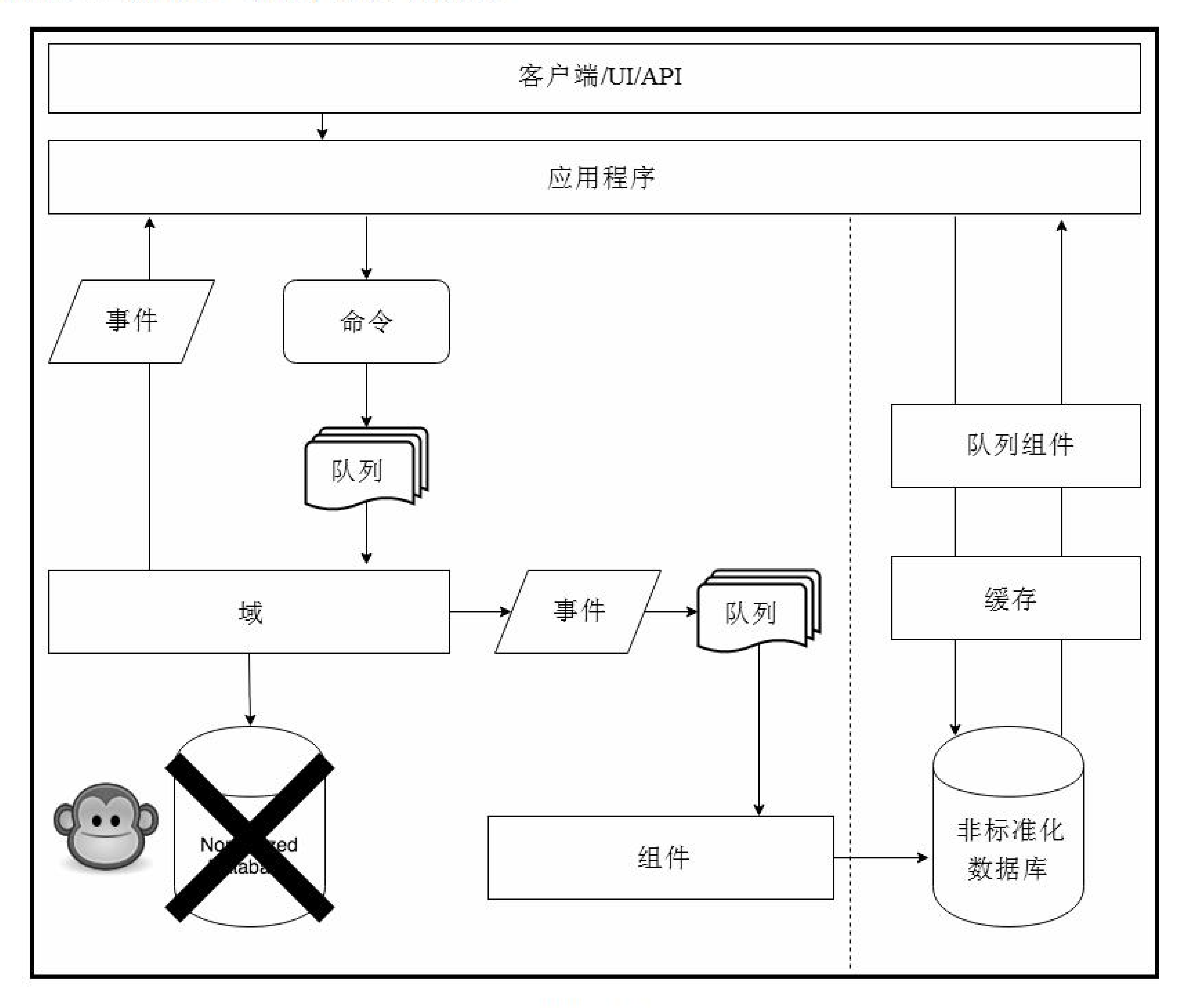


图 12.2

此处所产生的问题并不严重。其中，信息的写入过程将被中断，但读取过程并不会被打断。如果采用了事务型队列，未注册域使用的信息将返回到队列，直到 CommandStack 数据库实例被恢复。当在应用程序中配置了良好的监视系统后，可重新构建新的数据库实例，同时还可解决此类异常问题，并且不会对系统造成重大的影响。

注意，上述情形仅适用于 Chaos Monkey 测试。

Chaos Monkey 涵盖了以下测试分类。

- Chaos Gorilla：模拟全部可用区域中的不可用性。
- Conformity Monkey：关闭不符合最佳实践方案的实例。
- Doctor Monkey：执行性能检测（类似于 CPU）。
- Janitor Monkey：搜索未使用的资源并对其予以删除。
- Latency Monkey：在客户机-服务器通信中生成人为的延迟。
- Security Monkey：这将产生安全漏洞问题，例如经适当配置的非安全组。

12.9 本章小结

本章讨论了多种测试方案，并将所学的知识应用到实际案例中，包括最简单的单元测试，以及流行且功能强大的 Chaos Monkey 测试。

第 13 章将介绍监测、安全以及部署方案，其中涉及了许多新概念和策略方案，以进一步丰富微服务架构方面的知识。

第 13 章　安全监测和部署方案

前述章节讨论了应用程序所构建的逻辑内容，并使用了 DDD、内部设计模式，以及微服务间的通信设计模式；同时还考察了可扩展性、反模式以及良好的开发实践方案等内容。

本章将介绍如何将应用程序置入产品中，其中包含了较为广泛的概念，如下所示：

- 监测服务。
- 理解度量数值。
- 身份验证和授权服务。
- 单点登录。
- 安全数据。
- 人为因素。
- 攻击识别。
- API 网管。
- 持续集成/交付。
- 开发管线。
- 多服务实例/主机。
- 服务实例/主机。
- 服务实例/VM。
- 服务实例/容器。

不难发现，其中涉及大量的新内容需要我们进一步理解。

13.1　监测微服务

由于缺少相应的监测机制，某些微服务无法置入产品环境中。即使在可控环境下执行了相应的测试，也难以获得令人满意的性能结果。产品编码中包含了与可伸缩性、可用性以及应用程序的性能相关的内容。

缺少警告系统以及微服务的度量结果获取方式，常会导致较高的命中数量以及应用程序的覆写行为；抑或导致系统中的某些部分出现不稳定现象。其中，一些缺少检测的

不稳定性特征往往是最为危险的情况，且多为静默错误且难以检测。当发现问题时，为时已晚。

在某些场合下，由于开发团队并未真正理解应用程序的检测度量机制，或者并不知晓其采集方式，因而也会产生微服务不稳定这一类现象。

13.1.1　监测单一服务

总体来说，应用程序的检测机制并不复杂，但其中蕴含了分析和监测方面的知识。下面首先讨论单一服务器的监测，随后介绍多服务的监测。

监测机制主要涉及两方面的内容，即机器监测（测量应用程序服务器的可用性、健康程度以及性能）以及应用程序的监测（服务器机器是全功能型的，但应用程序则未必）。

在监测机制领域内，存在两种实现方式，即被动监测与主动监测。

- 主动监测是指受到监测的服务器将状态信息发送到监视工具。
- 被动监测则是指监测工具从服务器请求与机器或应用程序状态相关的信息。

图 13.1 显示了监测流程。

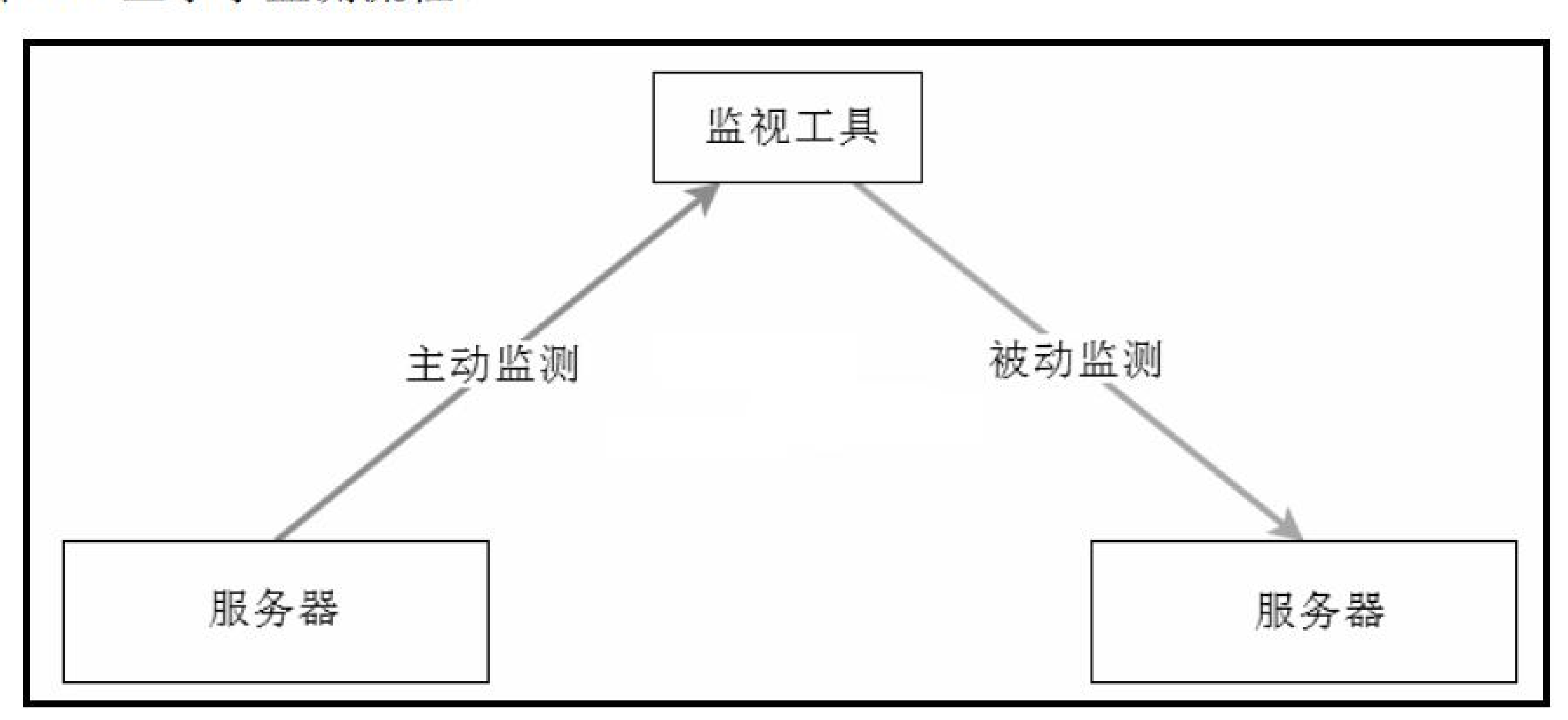


图 13.1

一种非常简单的方法是创建显示应用程序健康状况的端点，也称为健康检查。为了更好地理解这一内容，下面在 UsersService 微服务中创建该类型的端点。

在 UsersService 微服务的 app.go 文件中，向 initializeRoutes 方法中加入新的路径，如下所示：

```
func (a *App) initializeRoutes() {
      a.Router.HandleFunc("/all", a.getUsers).Methods("GET")
      a.Router.HandleFunc("/", a.createUser).Methods("POST")
```

```
	a.Router.HandleFunc("/{id:[0-9]+}", a.getUser).Methods("GET")
	a.Router.HandleFunc("/{id:[0-9]+}", a.updateUser).Methods("PUT")
	a.Router.HandleFunc("/{id:[0-9]+}", a.deleteUser).Methods("DELETE")
	a.Router.HandleFunc("/healthcheck", a.healthcheck).Methods("GET")
}
```

然后，将针对上述新路径创建处理程序。对应方法名和参数如下所示：

```
func (a *App) healthcheck(w http.ResponseWriter, r *http.Request) {
```

接下来，可定义一个变量并采集错误信息、搜索源自缓存的 Pool 连接，并筹备 Pool 连接的返回结果，如下所示：

```
var err error
c := a.Cache.Pool.Get()
defer c.Close()
```

下面执行第一次验证，也就是说，检测缓存是否处于活动状态，如下所示：

```
// Check Cache
_, err = c.Do("PING")
```

随后执行第二次验证，即检测数据库是否处于活动状态，如下所示：

```
// Check DB
err = a.DB.Ping()
```

如果其中的某个组件不可用，那么，将向当前监测工具返回一条错误信息，如下所示：

```
if err != nil {
	http.Error(w, "CRITICAL", http.StatusInternalServerError)
	return
}
```

如果应用程序组件中未包含任何错误，则返回一条消息并显示一切均正常，如下所示：

```
	w.Write([]byte("OK"))
	return
}
```

最终，健康监测处理程序包含以下内容：

```
func (a *App) healthcheck(w http.ResponseWriter, r *http.Request) {
	var err error
	c := a.Cache.Pool.Get()
	defer c.Close()
```

```
        // Check Cache
         , err = c.Do("PING")

        // Check DB
        err = a.DB.Ping()

        if err != nil {
                http.Error(w, "CRITICAL", http.StatusInternalServerError)
                return
        }

        w.Write([]byte("OK"))
        return
}
```

13.1.2　监测多项服务

在 13.1.1 节中，我们考察了单一服务监测的工作方式。但是，当谈及微服务架构时，其中将会涉及大量的服务器。那么，如何监测所有这些服务器？答案在于自动化。

通过人工方式，一般无法监测如此众多的服务器，我们需要借助工具的力量实现这一项任务，其中涉及多种有效的工具。相应地，一些工具以付费方式推出，而另一些工具则完全免费。这里需要特别关注一下 Nagios（对应网址为 https://www.nagios.org/downloads/nagioscore/）。Nagios 是一款灵活、可定制的工具，并可处理监测过程中的主要用例。其付费版本可极大地简化相关操作过程。

图 13.2 显示了 Nagios Core 服务器监测界面。

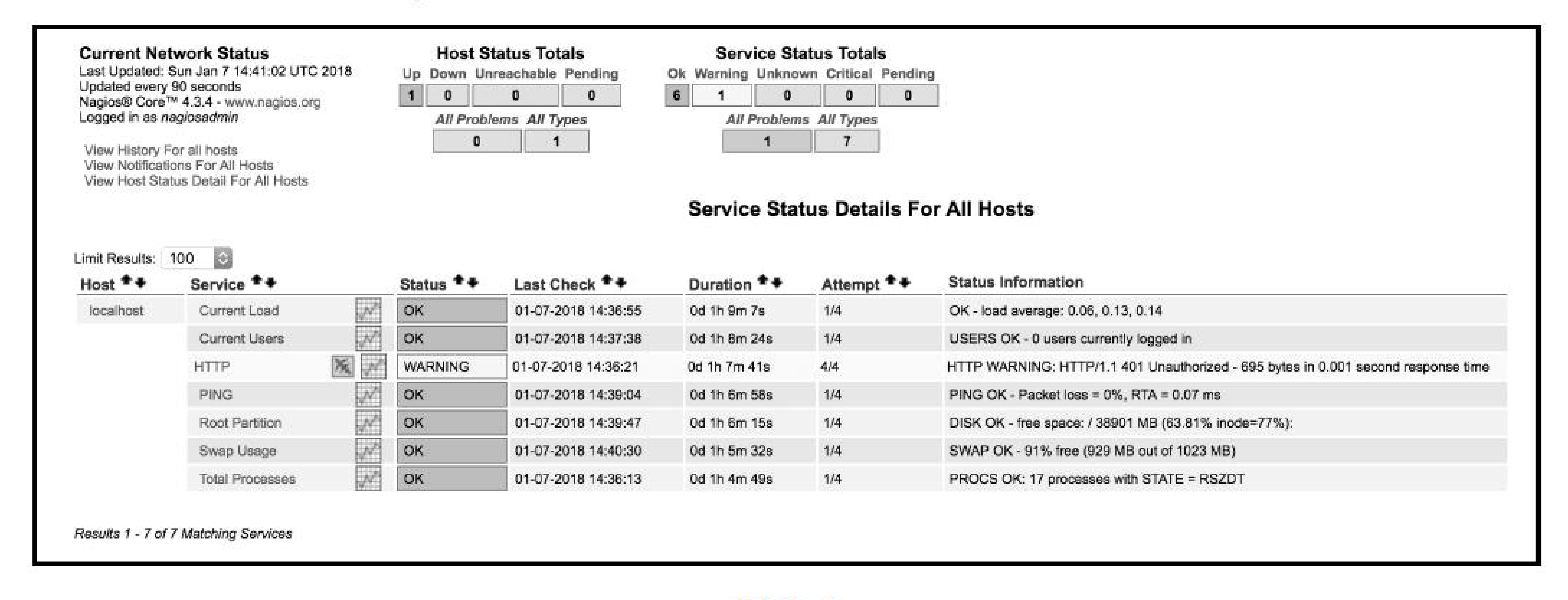


图 13.2

当以被动方式监测服务器时，图 13.3 展示了相关结果的描述方式。

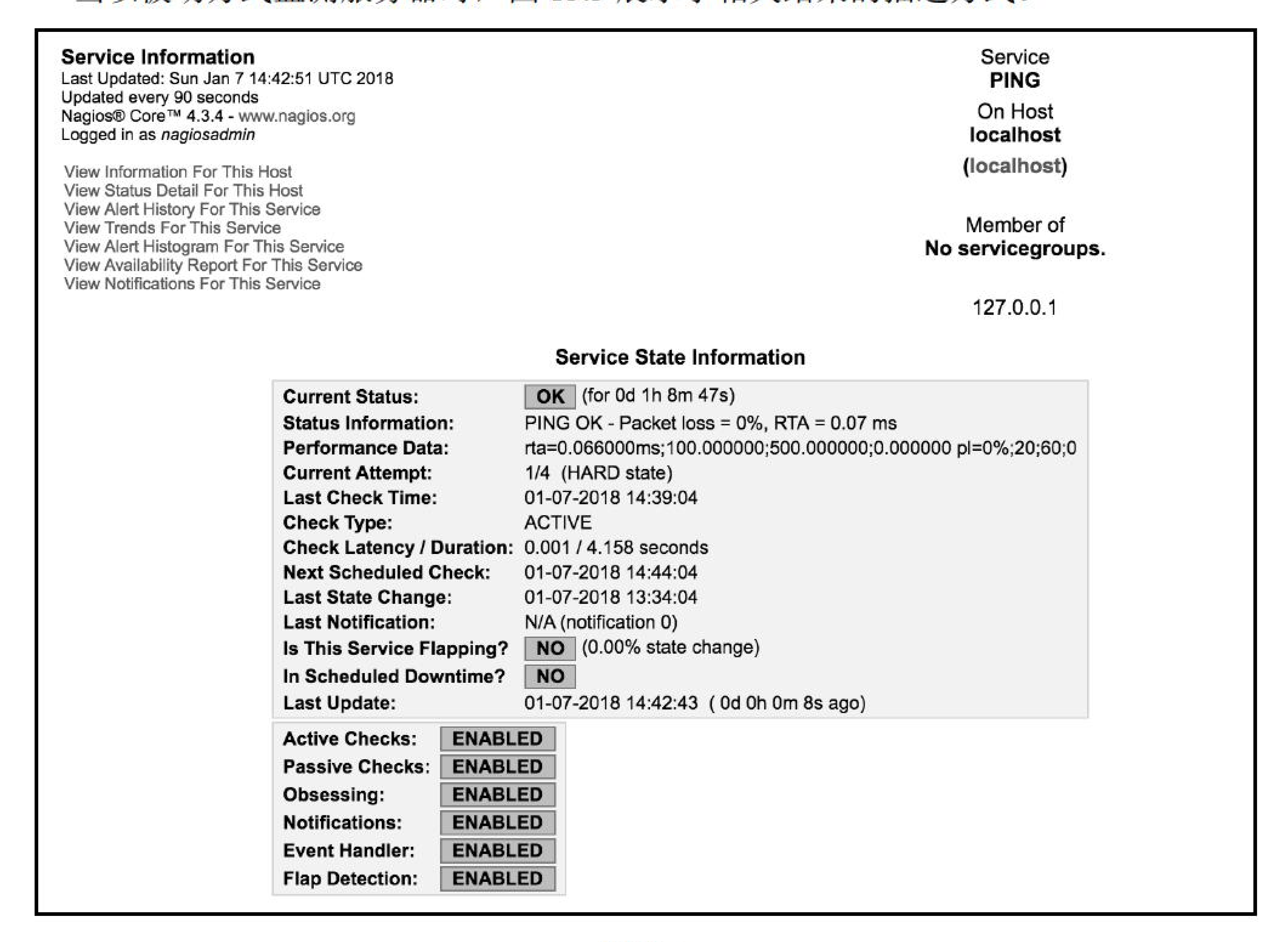


图 13.3

虽然 Nagios 的其他版本缺少应有的完整性和简单性，但是 Nagios Core 则非常灵活，且包含了丰富的文档内容。

13.1.3　查看日志

某些时候，软件开发人员更习惯于通过日志观察应用程序的行为。对于微服务和大规模应用程序来说，通过人工方式监测几乎是不可能的。

当手动方式查看日志异常复杂时，通过工具对日志进行分析，并以简单方式提供相关信息则变得十分有效。

对此，存在一些工具可实现上述目标，Splunk 则是其中的佼佼者。

Splunk 提供了免费版本，但用户额度以及日志数量（以 MB 计算）则较为有限。尽管如此，该版本对于用户熟悉 Splunk 的使用方式来说十分有用。总体而言，Splunk

提供了一个查询系统，可以对数百万行日志执行搜索操作。图 13.4 显示了 Splunk 的搜索示例。

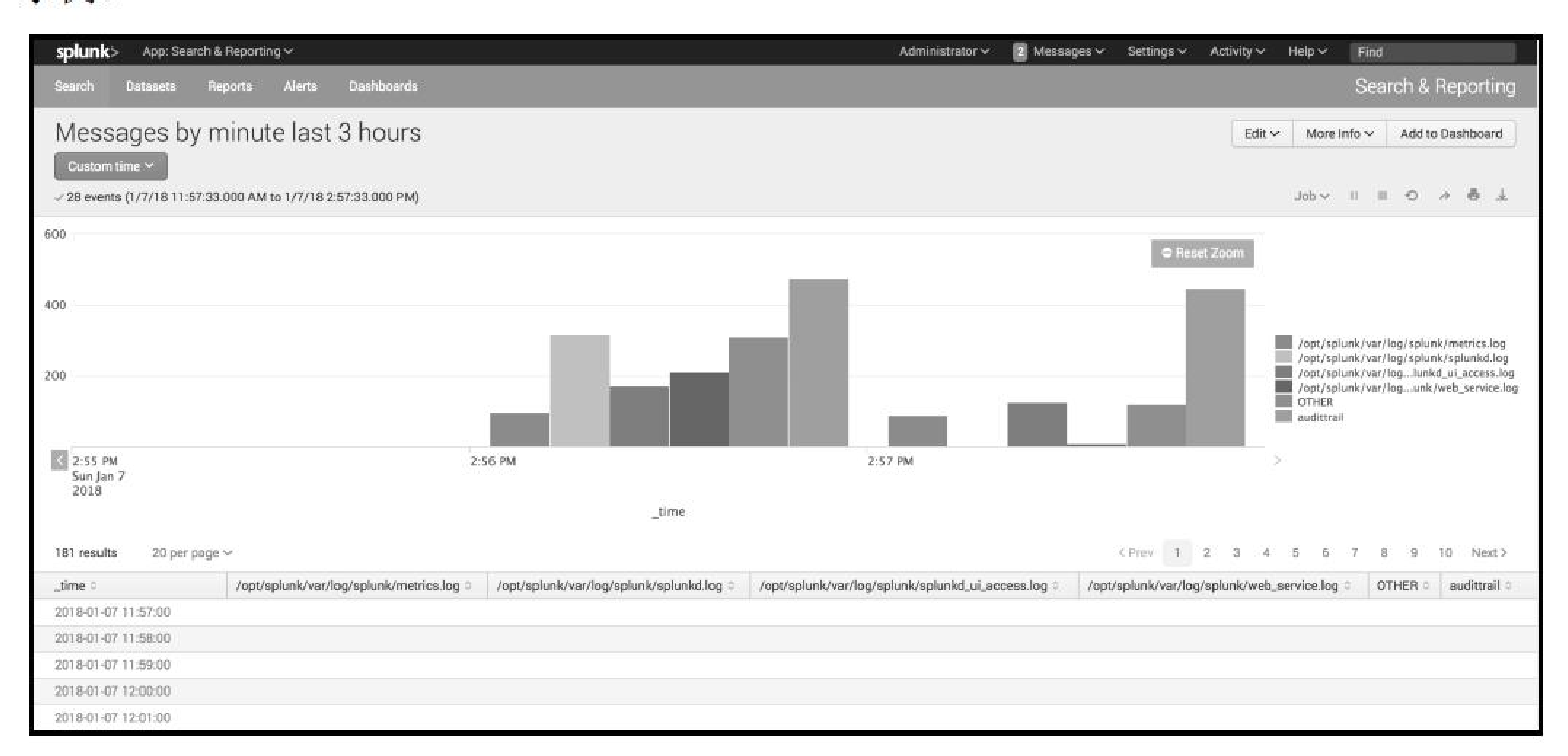


图 13.4

13.1.4　应用程序中的错误

当谈及软件开发时，往往会涉及应用程序错误这一话题。相应地，存在多种因素可导致应用程序出现错误。虽然存在较为广泛的测试覆盖以及良好的监测机制，但在某个关键点上，错误仍在所难免。

首先，读者不要对错误抱有畏惧心理。错误总会发生，但重要的是应对此类错误历史加以记录。跟踪历史记录与下列行为有关：故障的文档记录方式，以及如何快速恢复该文档，以对应用程序的恢复提供有效的帮助。

编写宕机文档似乎有些过时，但是，具有更新内容且编写良好的文档仍是最为高效的交流形式之一。除了生成历史数据之外，此类文档并不会依赖于某个人的特定记忆。

记录应用程序的工作方式是一种较好的习惯，但其缺陷也较为明显。Sphinx 则是一种较好的文档编写工具，该工具易于编辑，并提供了智能搜索功能。

Sphinx 能够通过动态搜索，并以集中化、索引化的 HTML 格式提供文档。另外，Sphinx 不与任何特定的文本编辑器绑定。

Sentry 则是另一个捕捉错误和保存历史记录的较好工具。通过简单的代码插入，所有严重的错误都可以报告给 Sentry。该工具的一个有趣之处是，除了维护历史记录之外，

还可提供错误的出现频率。

下面在 UsersService 微服务中尝试使用 Sentry。在 main.go 文件中，将添加下列导入语句，进而将故障报告于 Sentry。

```
import (
	"flag"
	"log"
	"os"
	"github.com/jmoiron/sqlx"
	  "github.com/lib/pq"
	raven "github.com/getsentry/raven-go"
)
```

接下来在 main.go 文件中定义 init 方法，在每次启动应用程序时，该方法于 main 方法之前运行。另外，该方法还定义了相关配置，以显示错误的发送位置。对应代码如下所示：

```
func init() {
	raven.SetDSN("<YOUR SENTRY ROUTE>")
}
```

出于演示目的，在 Sentry 配置完毕后，将编写一个新的处理程序并总会产生错误——该程序试图打开一个并不存在的文件。显然，我们需要在 app.go 文件中定义新的路径，如下所示：

```
func (a *App) initializeRoutes() {
	a.Router.HandleFunc("/all", a.getUsers).Methods("GET")
	a.Router.HandleFunc("/", a.createUser).Methods("POST")
	a.Router.HandleFunc("/{id:[0-9]+}", a.getUser).Methods("GET")
	a.Router.HandleFunc("/{id:[0-9]+}", a.updateUser).Methods("PUT")
	a.Router.HandleFunc("/{id:[0-9]+}", a.deleteUser).Methods("DELETE")
	a.Router.HandleFunc("/healthcheck", a.healthcheck).Methods("GET")
	a.Router.HandleFunc("/sentryerr", a.sentryerr).Methods("GET")
}
```

随后，在 app.go 文件中编写一个处理程序，并针对 Sentry 生成错误，如下所示：

```
func (a *App) sentryerr(w http.ResponseWriter, r *http.Request) {
```

由于尝试读取的文件并不存在，因而下列代码行将产生错误：

```
_, err := os.Open("filename.ext")
```

对此，捕捉错误并将其发送至 Sentry 中，同时在 HTTP 响应中返回一个错误。对应

代码如下所示：

```
if err != nil {
       raven.CaptureErrorAndWait(err, nil)
       http.Error(w, err.Error(), http.StatusInternalServerError)
       return
}
```

如果未出现任何错误，响应结果将显示为“OK”，如下所示：

```
       w.Write([]byte("OK"))
       return
}
```

最终，当前方法的完整内容如下所示：

```
func (a *App) sentryerr(w http.ResponseWriter, r *http.Request) {
        , err := os.Open("filename.ext")
       if err != nil {
              raven.CaptureErrorAndWait(err, nil)
              http.Error(w, err.Error(), http.StatusInternalServerError)
              return
       }
       w.Write([]byte("OK"))
       return
}
```

要在 Sentry 中生成错误，只需访问以/v1/users/sentryerr 结尾的 URL。图 13.5 中显示了 Sentry 中的错误结果。

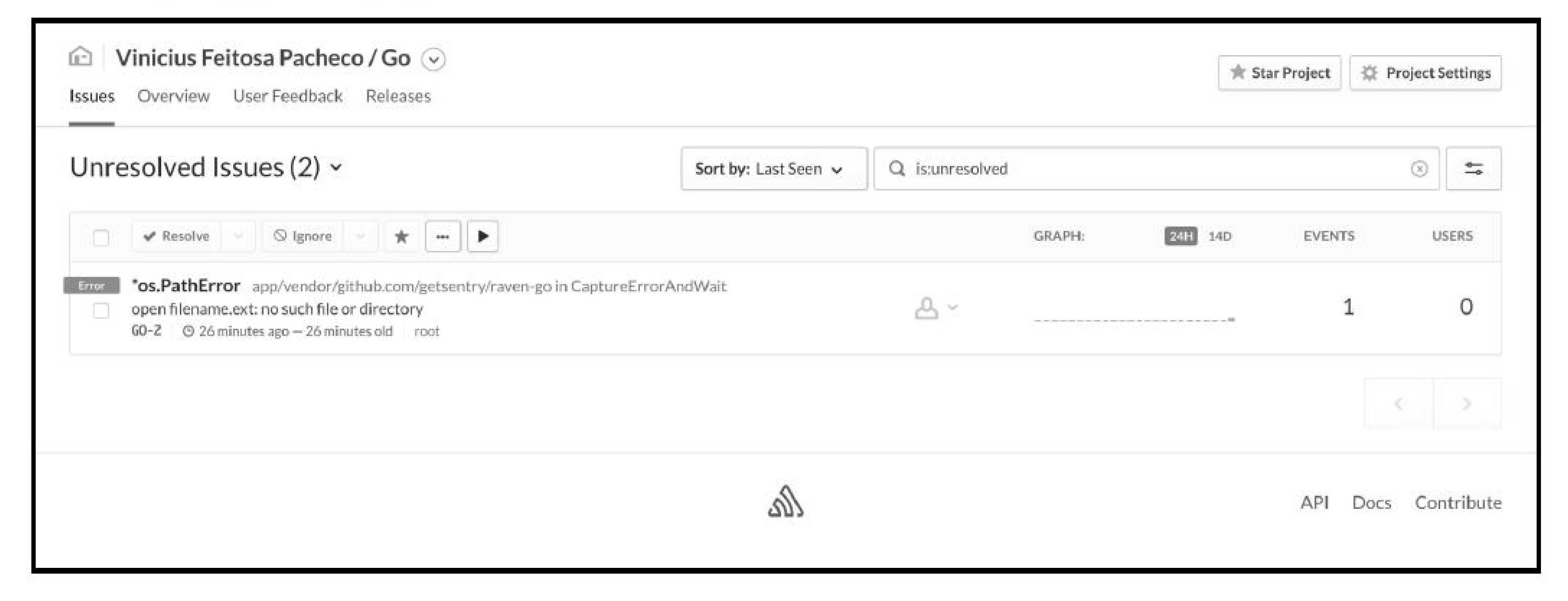


图 13.5

当单击 Sentry 中所报告的错误链接时，将会出现如图 13.6 所示的错误报告。



图 13.6

不难发现，其中包含了详细、完整的错误信息。如果重复调用产生故障的端点，Sentry 将识别相同的错误，并显示同一错误在一段时间内的出现次数，如图 13.7 所示。

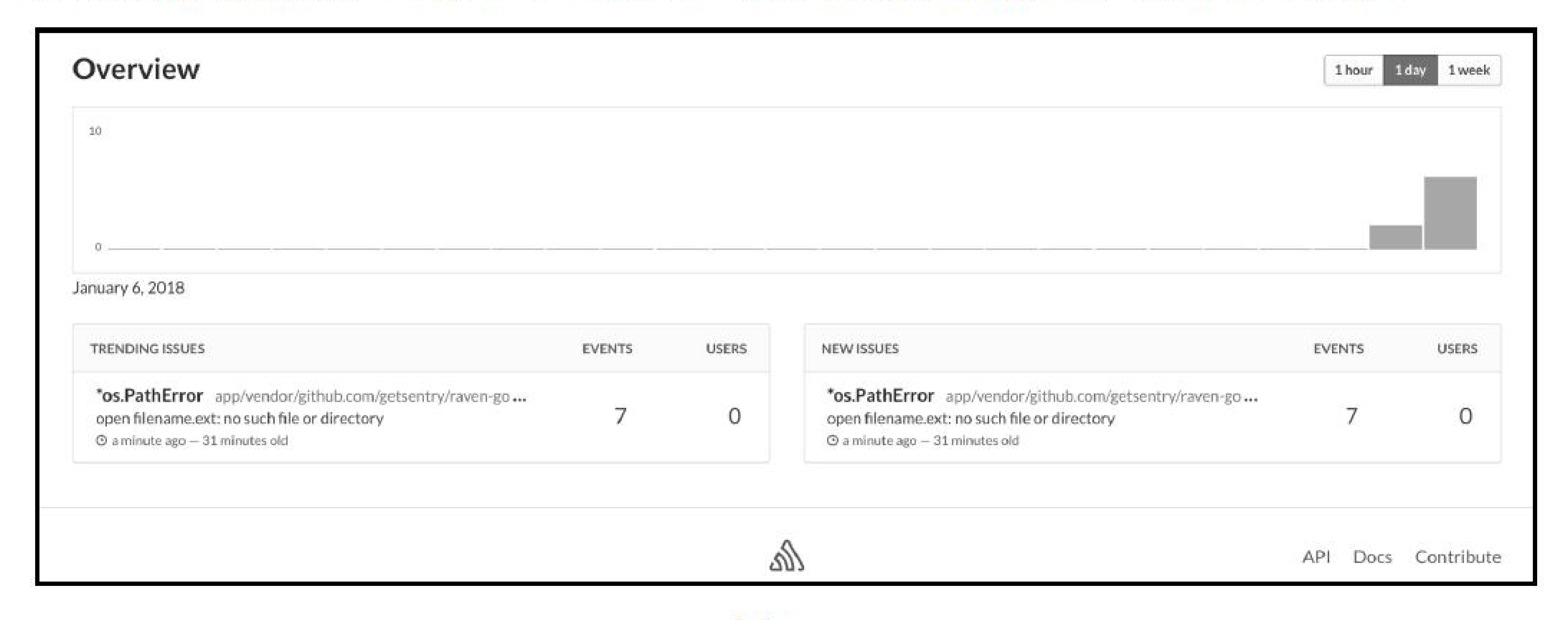


图 13.7

综上所述，跟踪应用程序中的错误是快速恢复系统的关键所在。

13.1.5　度量方法

度量标准十分有用，并可使我们更加深入地理解某种事物。常见的度量方法是考察

平均值，但当处理微服务时，这将是一个很大的错误。

度量结果的平均值可能会掩盖某些异常情况。考察下列数值：

```
  4 threads and 50 connections
  Thread Stats  Avg      Stdev    Max   +/- Stdev
    Latency    28.68ms  67.34ms  1.55s   98.58%
    Req/Sec   554.47   154.04    1.06k   73.50%
  22085 requests in 10.01s, 3.45MB read
  Socket errors: connect 0, read 4, write 0, timeout 0
Requests/sec:   2206.44
Transfer/sec:    353.28KB
```

对于延迟平均值来说，一切均正常且平均值为 28.68 毫秒。然而，当查看最大延迟时，问题便会出现。其中，最大时间值是 1.55 秒。这意味着一个请求几乎需要两秒钟才能完成。

这种数字上的异常有助于我们更好地理解工具反馈给我们的度量结果。此外，我们还可以通过这一类数值制定更好的可伸缩性策略。

13.2　安全问题

注意，存在多种方式可使应用程序遭受攻击。因此，了解如何保护微服务十分重要，进而可避免软件在几秒钟内被破坏。

13.2.1　理解 JWT

当与 API 协同工作时，需要考虑数据传输的安全性，特别是每个用户所持有的权限级别。对此，存在多种方式可实现这一任务。考虑到安全性和实现的简单性，JWT（JSON Web 令牌）表现得较为突出。

JWT 是一种可以通过 URL、POST 或 HTTP 报头发送的数据传输系统。这一类信息采用了数字签名，例如，使用 HMAC 算法或使用 RSA 算法的公钥/私钥签名。

JWT 的结构包含 3 个部分，并由点号予以分隔。这 3 部分内容分别是数据头、负载以及签名。下列代码显示了 Go 语言中 JWT 令牌的创建和读取过程。类似于其他代码，首先是数据包声明以及导入语句，如下所示：

```
package main
```

```
import (
	"fmt"
	"log"
	"time"
	jwt "github.com/dgrijalva/jwt-go"
)
```

在上述代码中，声明了 jwt-go 包，该数据包提供了令牌创建、使用所需的一切内容。下面声明 struct 方法，该方法也是令牌中的一部分内容。需要注意的是，struct 方法表示为发送数据与默认 JWT 结构间的组合结果。对应代码如下所示：

```
type MyCustomClaims struct {
	UserID int `json:"ID"`
	Name string `json:"name"`
	Rule string `json:"rule"`
	jwt.StandardClaims
}
```

在 struct 方法定义完毕后，下面考察 createToken 函数。首先需要声明函数名，以及令牌签名密钥，如下所示：

```
func createToken() string {
	mySigningKey := []byte("AllYourBase")
```

下面利用令牌负载数据构建 struct 方法，如下所示：

```
// Create the Claims
claims := MyCustomClaims{
	1,
	"Vinicius Pacheco",
	"Admin",
	jwt.StandardClaims{
		ExpiresAt: time.Now().Add(time.Hour * 72).Unix(),
		Issuer:    "Localhost",
		IssuedAt:  time.Now().Unix(),
	},
}
```

利用所创建的 struct 方法，我们将把相同内容传递至 JWT，随后利用签名字符串构建令牌。如果一切顺利，将返回相应的令牌，对应代码如下所示：

```
	token := jwt.NewWithClaims(jwt.SigningMethodHS256, claims)
	ss, err := token.SignedString(mySigningKey)
```

```
        if err != nil {
                log.Fatal(err.Error())
        }
        return ss
}
```

接下来定义 readToken 函数。首先需要声明函数名以及所接收的参数，如下所示：

```
func readToken(tokenString string) {
```

随后执行令牌解析过程，如下所示：

```
token, err := jwt.ParseWithClaims(
        tokenString,
        &amp;MyCustomClaims{},
        func(token *jwt.Token) (interface{}, error) {
                return []byte("AllYourBase"), nil
        },
)
```

在令牌解析过程执行完毕后，我们验证令牌是否完整有效。如果一切正常，将显示令牌值，如下所示：

```
        if claims, ok := token.Claims.(*MyCustomClaims); ok &amp;&amp;
token.Valid {
                fmt.Printf(
                        "%v %v %v %v\n",
                        claims.UserID,
                        claims.Name,
                        claims.Rule,
                        claims.StandardClaims.ExpiresAt,
                )
        } else {
            fmt.Println(err)
        }
}
```

除此之外，还需要定义 main 函数，以切实有效地执行上述处理过程，如下所示：

```
func main() {
        token := createToken()
        fmt.Println(token)
        readToken(token)
}
```

JWT 的简洁性令人印象深刻。通过这种技术实现，可构建一系列的身份验证和授权实践方案。

13.2.2　单点登录

单点登录（SSO）的应用十分广泛，我们很可能尚未意识到这一点。一个很好的例子是，我们使用某项服务的用户账户登录到另一个完全不同的服务。

当采用 JWT 时，即可构建提供 SSO 的应用程序。图 13.8 显示了基于 JWT 的 SSO 常见行为。

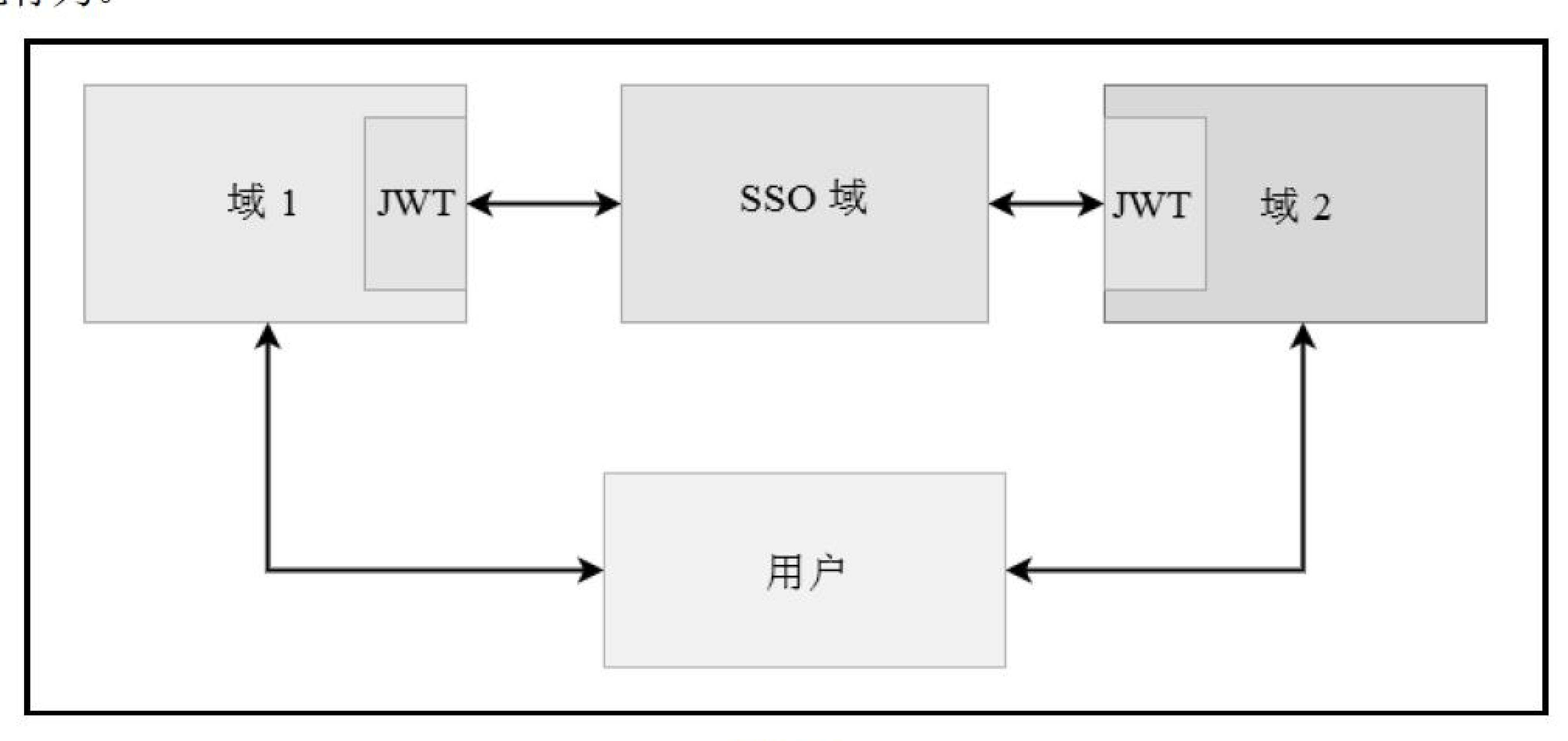


图 13.8

通过 SSO，可向应用程序提供更多的动态内容，除了产品自身之外，还可进一步提供验证和授权服务。

需要了解的重要一点是，需要针对身份验证中的专用微服务进行优化——在很大程度上，它有可能成为系统的瓶颈。为了减少身份验证微服务可能产生的冲突，通常可将其划分为两部分内容。第一部分负责令牌的验证，第二部分则负责令牌的生成操作。

图 13.9 显示了两个不同的流程，分别用于创建令牌，以及验证所接收的令牌是否有效。

图 13.9 中使用了基于插件的 Nginx，并执行了令牌的验证工作。此类策略可有效地降低逻辑层所承受的压力，从而减少瓶颈出现的概率。

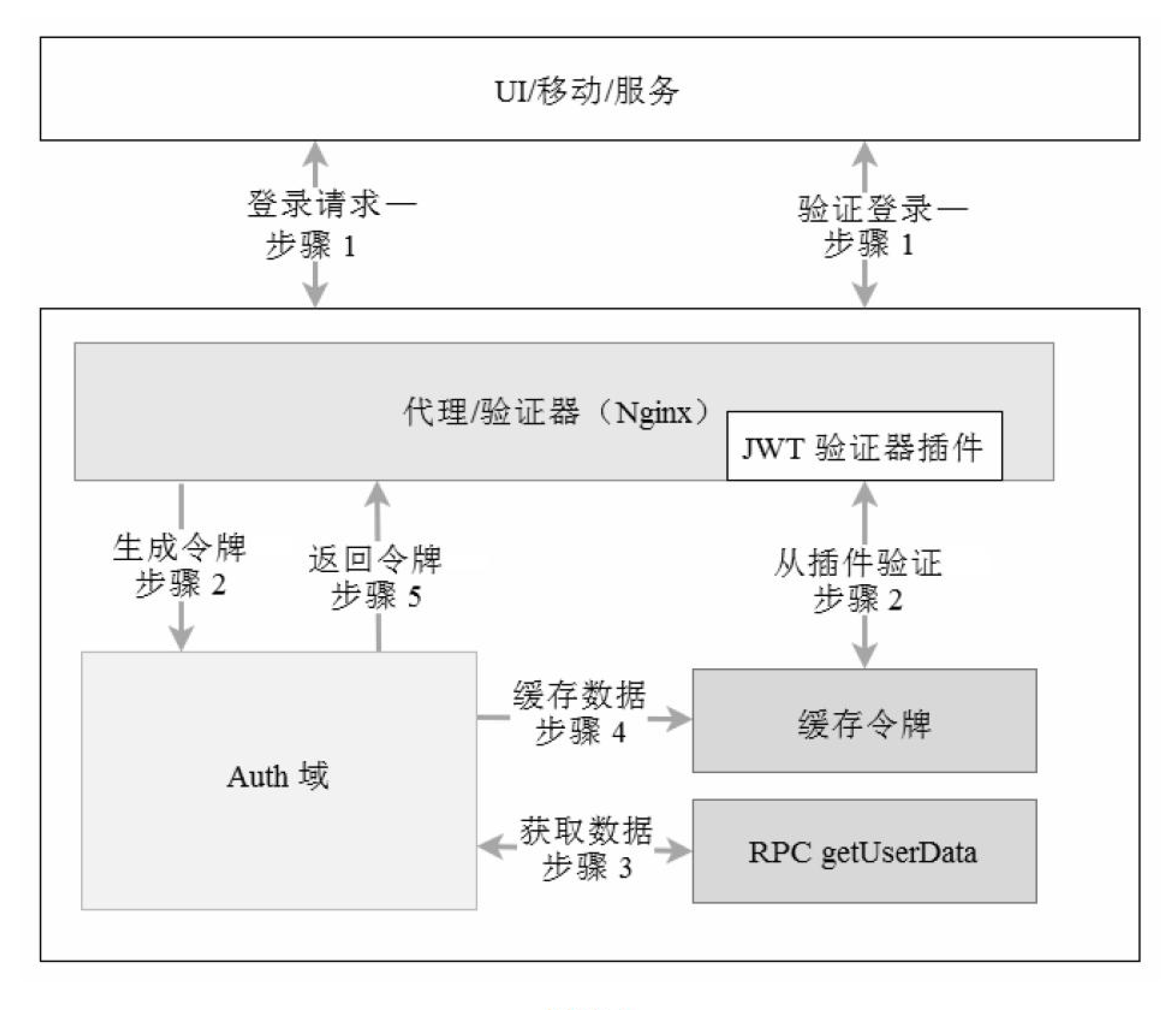


图 13.9

13.2.3　数据安全

安全地隔离数据层是每个开发团队都在考虑的事情。即使数据被防火墙或者其他类型的策略安全地隔离，仍会存在一些敏感数据无法实现正确的存储。

众所周知，密码即是一类敏感数据。但是，许多开发团队仍对数据存储采取了可还原加密，而非单向哈希值。这里，可还原的密码可视为一种安全漏洞。

对于数据来说，另一种较好的实践方案是避免使用顺序数字 ID；否则，用户数据将极易被识别或窃取。对于应用程序来说，在数据库中使用哈希值或跳跃式数字则更加安全。如果可能的话，建议使用哈希值作为 ID。

另外，还可采用 HTTPS，甚至低于 API 级别。仅针对应用程序的外部通信层使用 HTTPS 的应用程序是一种十分常见的情形。

13.2.4　预防恶意攻击——识别攻击行为

基于微服务架构，最常见的攻击类型是分布式拒绝服务（DDoS）。鉴于其简单性，且涉及微服务的高计算能力（计算资源有限，尤其是应用程序遭受攻击时），因而这种类型的攻击行为较为常见。

首先需要对攻击行为进行识别，对此存在一类特定的模式。第一点需要注意的是人为因素，在这种情况下，需要计算微服务的调用事件。无论是否遭受到攻击，调用速度可视为一种较好的标识。

一旦请求被确认并通过验证，建议使用以下缓解策略：

- 构建最小化微服务间依赖关系的架构。
- 理解服务彼此间的使用方式，以及服务的调用方式。例如，可限制对象的批处理或者请求执行的大小。
- 从后端服务向应用程序防火墙提供反馈。就 API 调用特性的使用来说，这将传递与此相关的附加信息，这些特性是无法识别的。
- 监控缓存对象的缺失。大容量可能意味着缓存没有正确地配置。
- 采用自定义弹性标准，例如断路器或超时方案。

这些做法并不一定总能够阻止攻击，但肯定会将副作用降至最低。

13.2.5　拦截器

拦截器的目的是在代码流中设置一个类，负责收集/操作通过 HTTP 请求传递的信息，例如 HTTP 代理提供的认证头、URL 和证书。图 13.10 展示了这一概念。

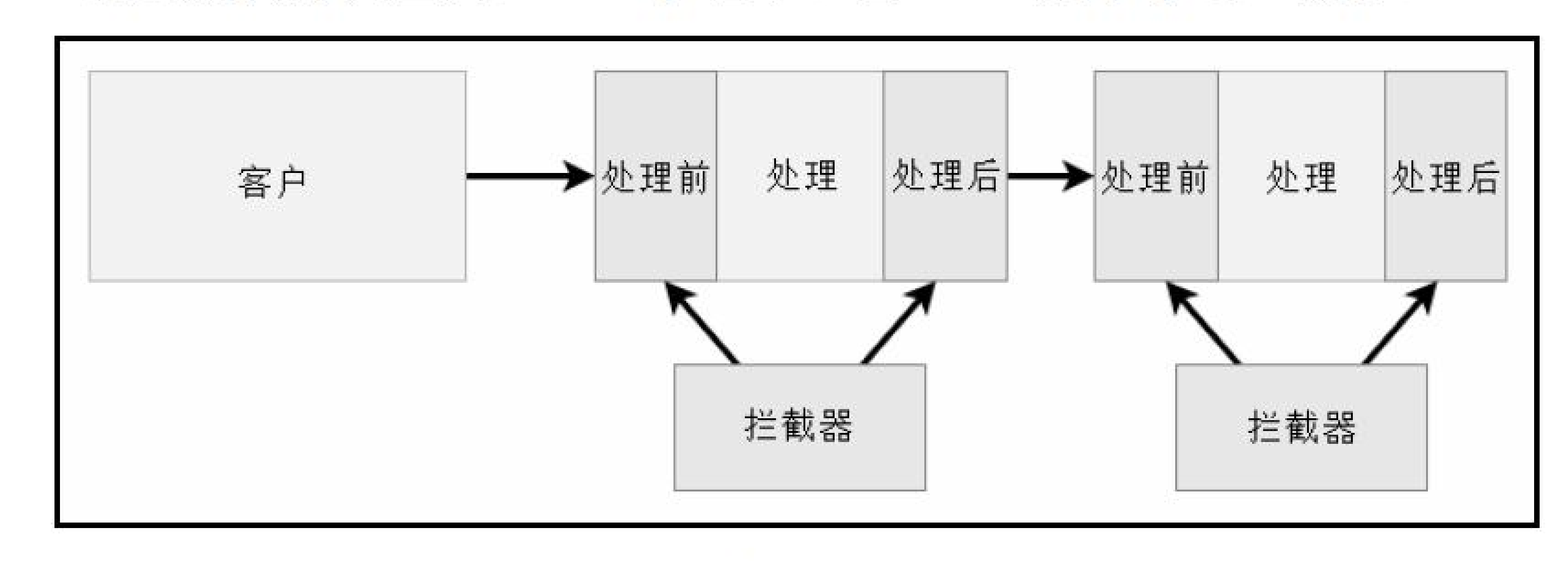


图 13.10

拦截器可在处理过程调用前后予以添加，对应处理过程负责处理在应用程序内声明的逻辑内容。相应地，拦截器包含了相关代码，并可视作一种安全保障。

未纳入拦截器处理策略中的任何数据，都将导致在应用程序中执行某些操作。此类操作包括简单的日志，甚至是拦截微服务的节点操作，进而维护应用程序的整体完整性。需要注意的是，这一类技术无法防止套接字耗尽这一现象的出现。

13.2.6 容器

Web 容器是与 Java servlet 交互的 Web 服务器的组件。Web 容器标准非常有用，它提供了一系列的工具来提高应用程序的安全性，并有助于实现相关规则和访问策略的中心化操作。下列示例显示了 Web 容器的操作方式：

- 支持安全套接字层（SSL）/传输层安全（TLS），并可对 HTTP 代理和服务器间的交换数据进行加密。
- 负责配置交互 TLS。
- 通过与用户角色关联的安全约束来限制对 Web 资源的访问。

Web 容器位于应用程序之上，并作为第一级安全服务，特别是当涉及消息加密时。图 13.11 显示了 Web 容器相对于应用程序的定位。

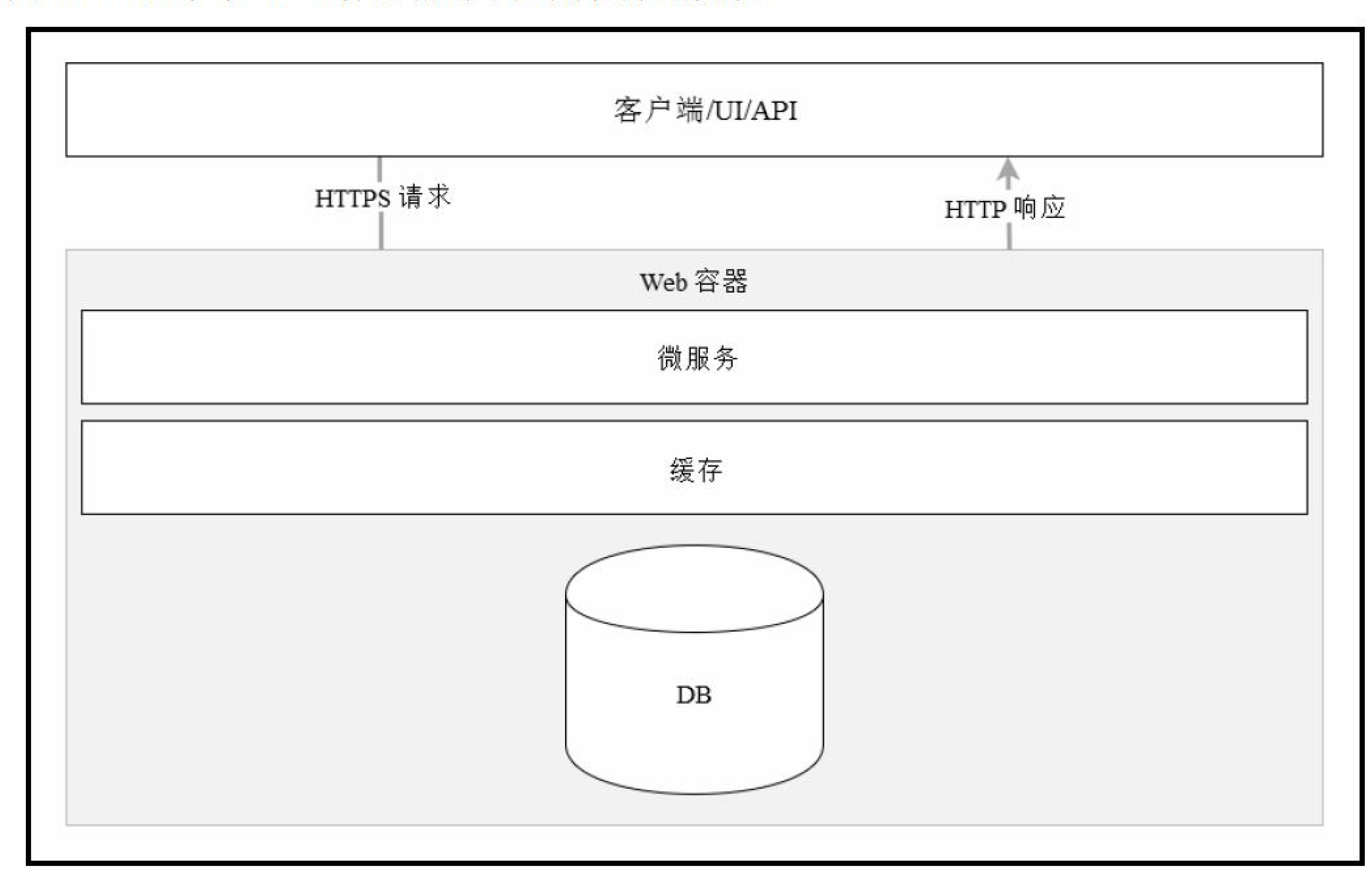


图 13.11

13.2.7　API 网关

如果将微服务比作一个庞大的管弦乐队，那么，网关 API 则担任了指挥这一角色。网管模式负责设置微服务。

API 网关位于微服务之上，网关的优势主要体现在优化后的端点，以及集中式的中间件功能。

假设应用程序采用了微服务架构，并包含 3 种不同的客户端类型，即 Web 前端、移动应用程序以及应用程序外部的另一项服务。每个客户端都希望对各自的请求有不同的响应。如果缺少网关，每个客户端直接知晓负责传递信息的微服务及其操控方式。然而，API 网关所饰演的角色可以优化端点和处理响应。

集中式中间件功能意味着安全级别、权限级别、身份验证级别，以及其他位于网关级别的验证行为。

图 13.12 显示了相对于微服务的 API 网关位置。

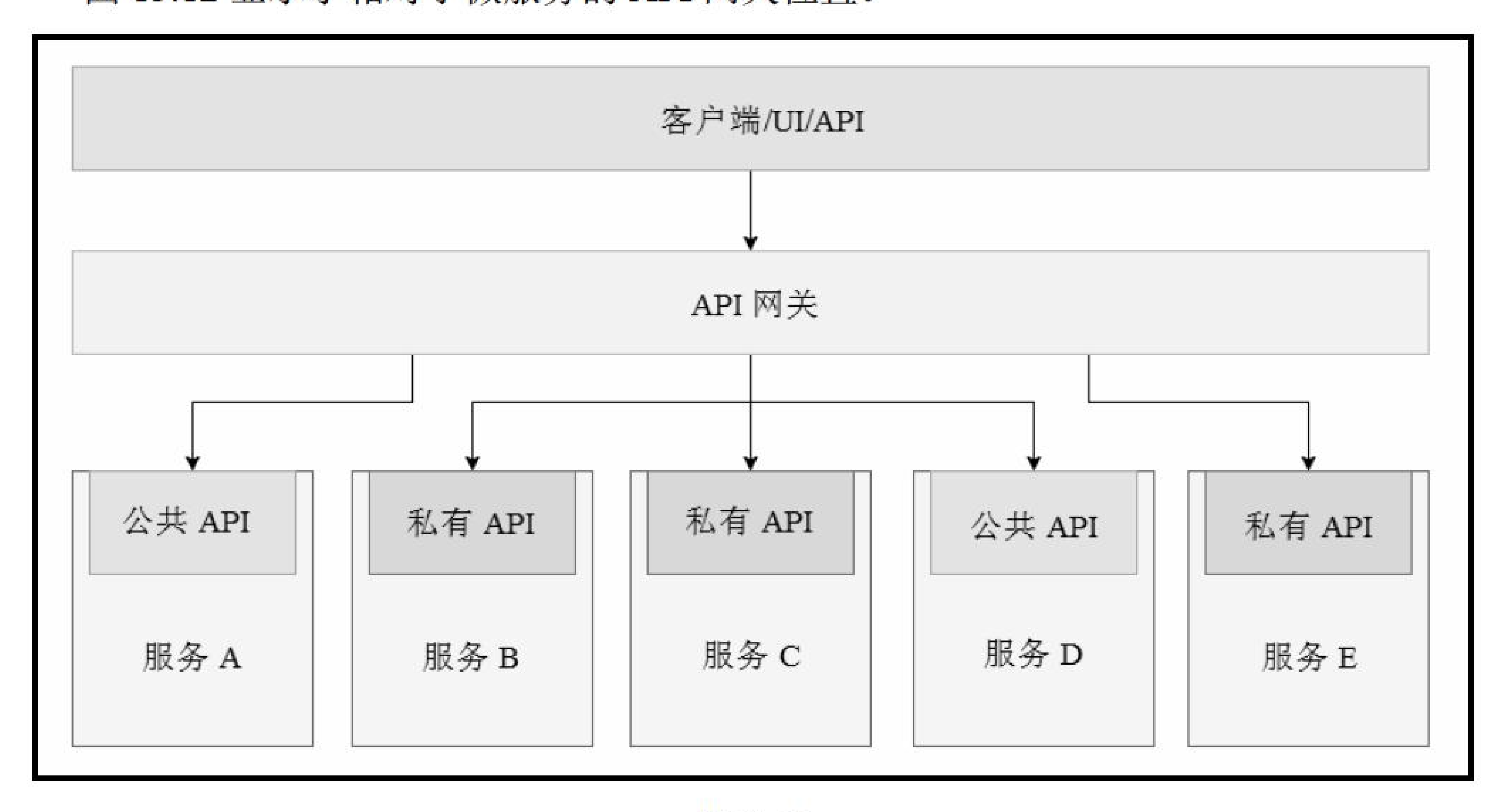


图 13.12

13.3　部　　署

前述内容讨论了应用程序的工作状态，以及安全项针对微服务的工作方式。本节在

微服务架构的基础上介绍部署模式。

13.3.1　持续集成和持续交付/持续部署

作为一种实践方案，持续集成是指开发人员尽可能频繁地将更改返回至主分支中。开发人员做出的更改是通过创建编译，并针对构建结果运行自动化测试加以验证的。在执行持续集成处理这一过程中，当开发人员等待新特性的发布，以合并在发布分支中的更改内容时，一些常见问题一般是可以避免的。

持续集成非常重视自动化测试，进而检测当新的提交内容被集成到核心业务中时，新特性是否存在问题。

持续交付可视作持续集成的扩展，以确保能够以敏捷和可持续的方式快速向客户发布新特性。这意味着，除了自动化测试之外，还实现了发布过程的自动化，用户可以通过单击按钮随时部署应用程序。

从理论上讲，通过持续交付，可以决定每天、每周、每两周发布的版本，或者任何适合业务需求的版本。如果打算真实感受持续交付所带来的好处，应尽早发送产品特性以确保小批量交付。这样，当出现问题时，即可方便地对其加以解决。需要注意的是，为产品发送应用程序版本的最终过程依赖于人工干预，如图 13.13 所示。

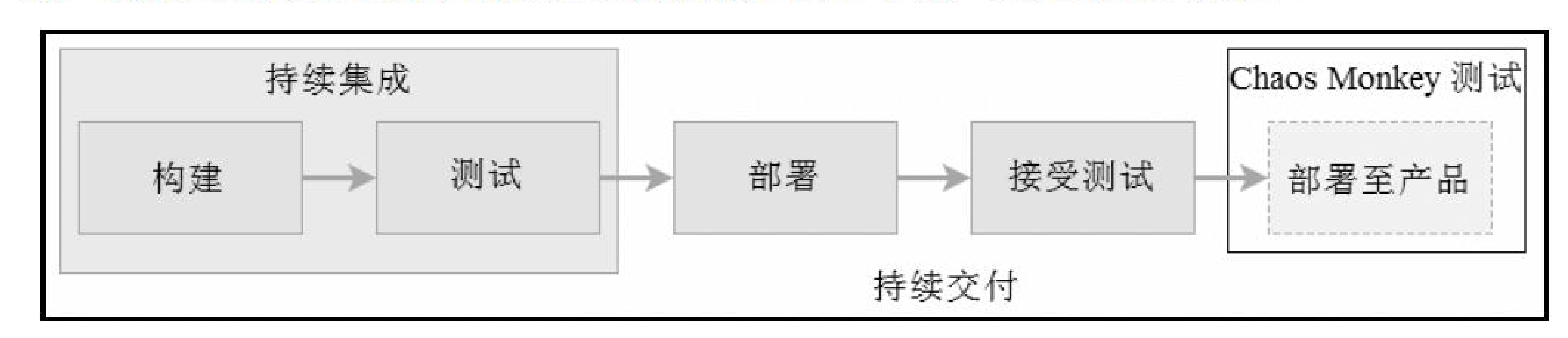


图 13.13

持续部署比持续交付更进一步。据此，产品管线中每个步骤中的每项变化都将发布于客户，其间不存在人为干预；另外，无效测试将会阻止新的变化内容进入至产品实现中。

持续部署是加速客户反馈循环的一种较好的方法。鉴于不存在特定的发布日期，新功能的交付不涉及技术结构方面的规划。开发人员可以专注于构建软件，并且在开发结束后的几分钟内就可以在生产环境中看到其工作成果。当然，实现这一发布模型需要较高的自动化测试覆盖率，以及较高的开发团队成熟度。

与持续集成不同，在该模型中，在向应用程序发布新版本时，将不存在人为干预，如图 13.14 所示。

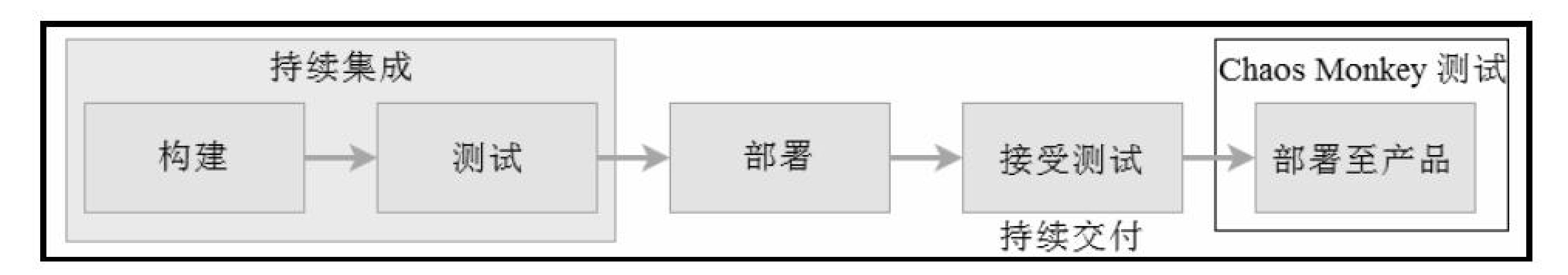


图 13.14

13.3.2　蓝/绿部署模式和 Canary 发布

从概念上讲，蓝/绿部署模式十分简单，并可针对产品高效地发送新的软件版本。对于部署蓝/绿模式的实现，具有微服务架构的 API 网关可以起到很大的帮助作用。

在单体应用程序中，一般来说，鉴于每次都需要发布软件的新版本，同时需要再次运行所有的单体部署，因而部署过程整体上十分缓慢。随着单体程序不断增长且越发庞大，整体速度将变得越发缓慢，问题也会随之出现。对于微服务来讲，我们可以多次部署单个组件。鉴于发送到生产环境的应用程序线程自身的大小，整个过程通常较快且易于实现。

在针对产品发布新的微服务版本时，采用了蓝/绿模式。例如，如果希望在版本 1.0.0 的基础上发布 1.0.1 版本，API 网关知晓这两个组件的位置，然后提供一个接口将传输过程从以前的版本切换到新的版本。图 13.15 描述了 API 网关的工作方式，并通过蓝/绿模式对版本进行了适当调整。

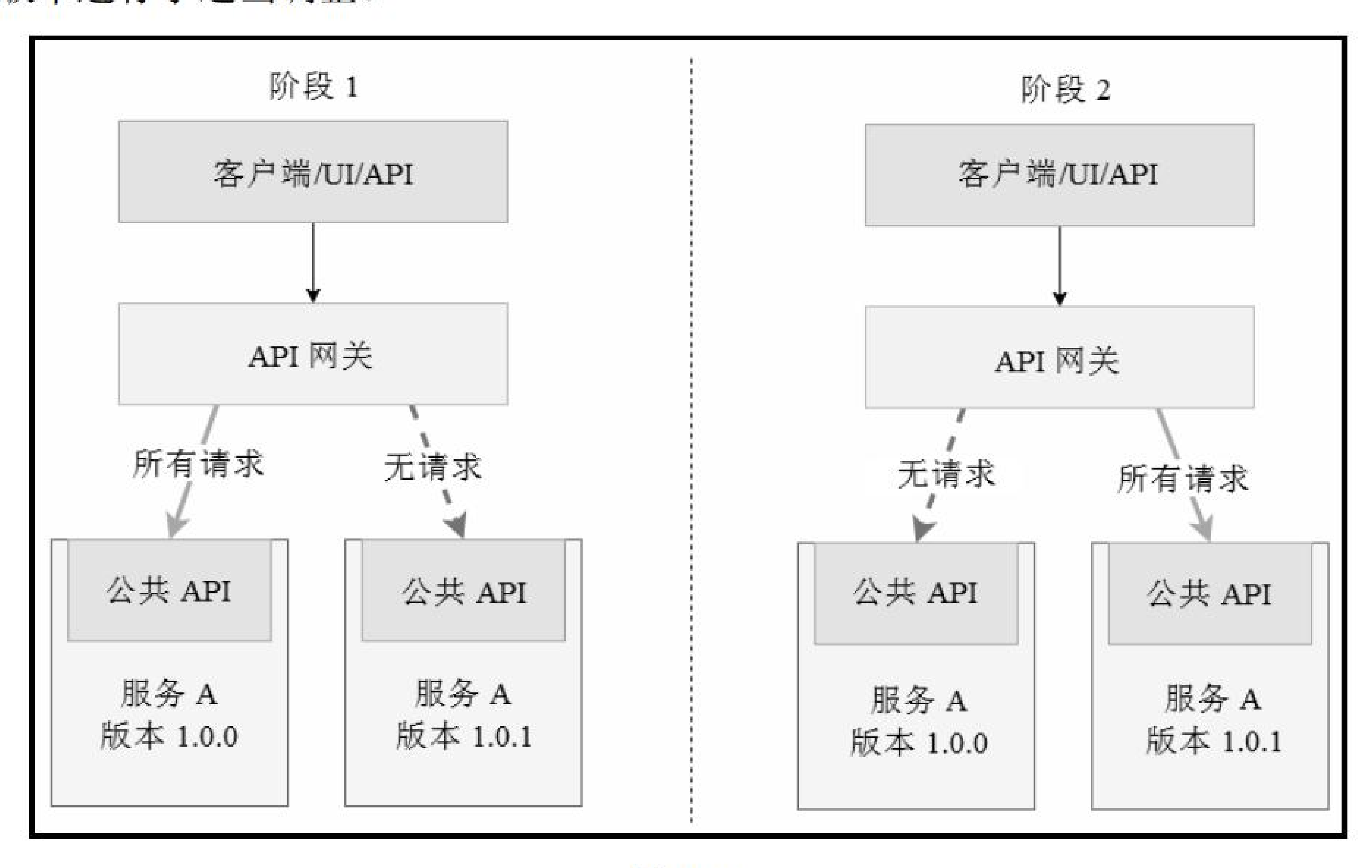


图 13.15

但是，某些部署可能会给应用程序的业务带来较大的风险。在蓝/绿模式中，版本的整体变化通常并不适宜。在运行部署时，应以一种受控的方式将请求定向到新版本的软件中。

针对于此，存在一种称之为 Canary 发布的模式。该处理过程类似于蓝/绿模式，二者间的差异是由于对新版本请求的逐步重定向而造成的。

请求控制可直接在网关上执行。通过这一方式，如果出现错误，对新版本发布的影响将会小得多。图 13.16 显示了这一处理过程。

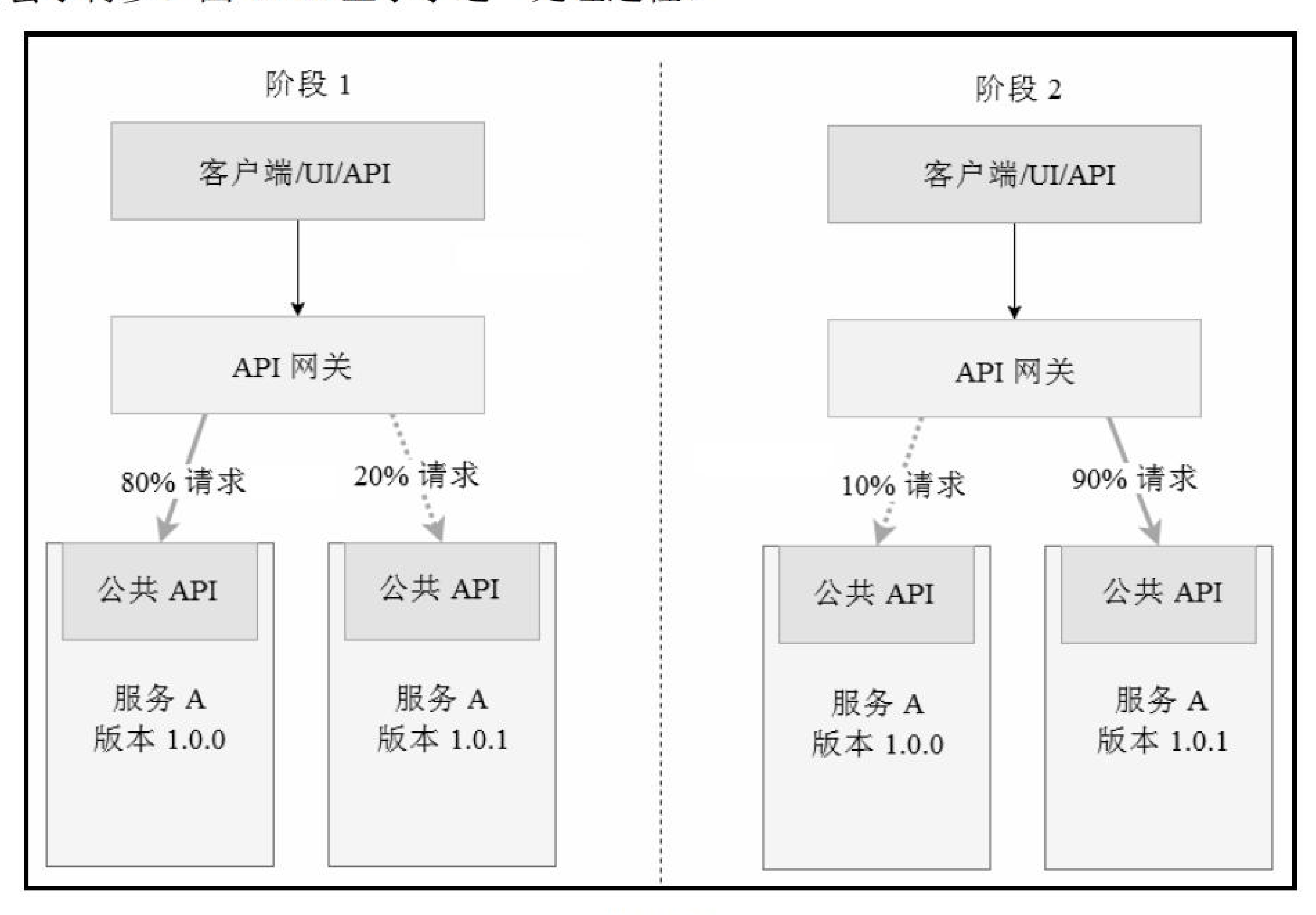


图 13.16

对于重定向到应用程序新版本的请求数量，这一过程与开发团队的信心有直接关系。该过程是以渐进方式进行的，且没有预先确定的时间，最终目标是将 100%的请求转移到应用程序的新版本中。

13.3.3　每台主机包含多个服务实例

通过上述模式，可提供一个或多个物理或虚拟主机，同时在其上运行多个微服务实例，这可视作是一种传统的应用程序部署方案。每个微服务实例在一台或多台主机上的已知端口上运行。

图 13.17 显示了该模式的结构。

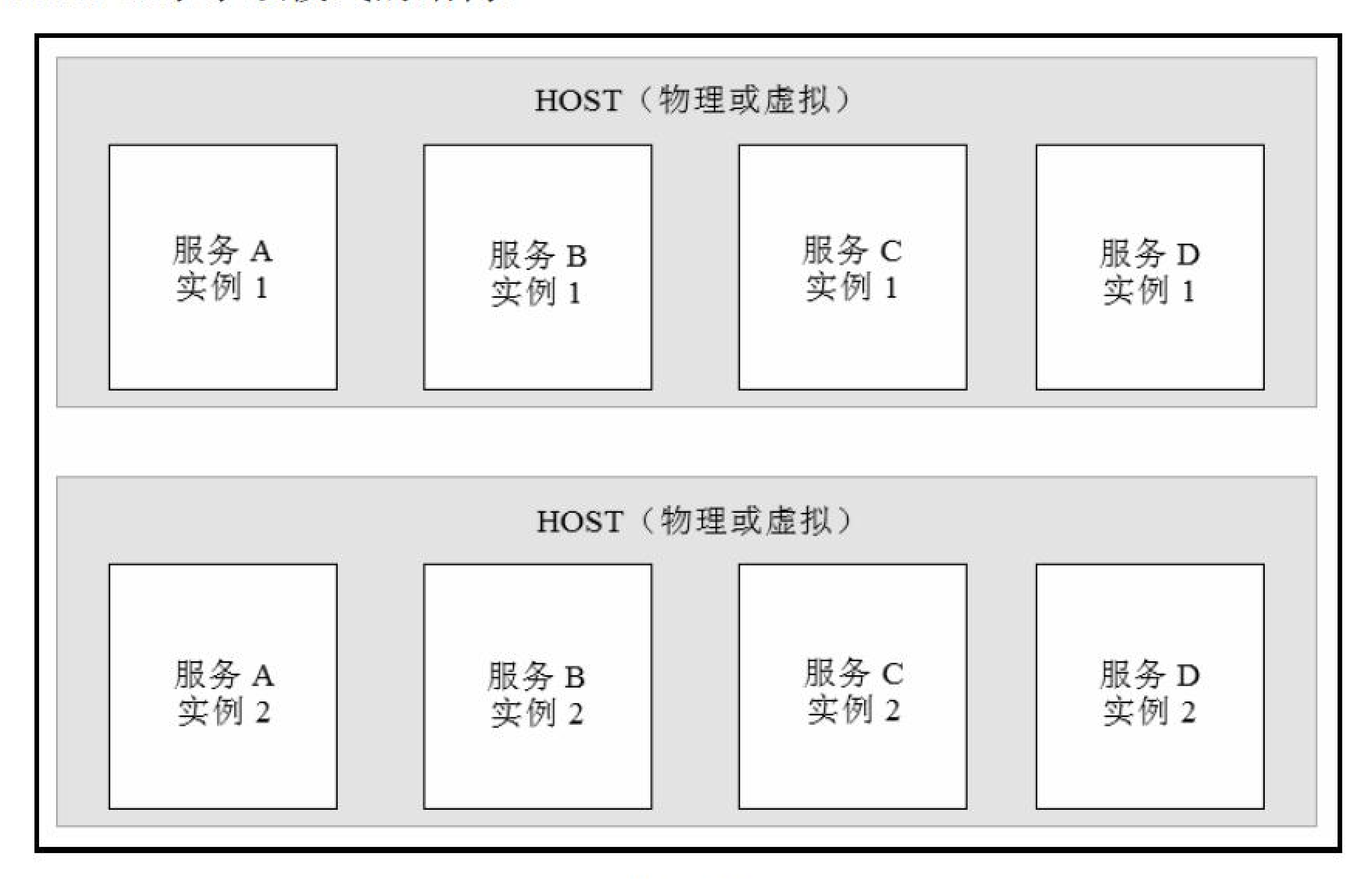


图 13.17

每台主机上的多个服务实例包含了自身的优缺点。其中，最主要的优点之一是可以高效地利用资源，多个服务实例可共享服务器及其操作系统；而微服务实例的快速部署则是该模式的另一个优点。

尽管如此，该模式中的实例也包含了某些较为显著的缺陷。例如，除非每个服务实例都是独立的处理流程，否则服务实例之间几乎不存在隔离。如果实例没有在不同的过程中被分离，那么，某个错误实例可能会危及整个过程，并且不可能对实例进行单独的监视。

13.3.4　每台主机的服务实例

当采用服务实例/主机这一模式时，将在其自身的主机上以隔离的方式运行每个微服务实例。相应地，该模式中存在两种不同的模型，即每个 VM 的服务实例和每个容器的服务实例。

1．每个 VM 的服务实例

其中，我们将每个微服务作为虚拟机（VM）的映像。每个微服务实例表示为一个 VM，执行该 VM 可使微服务处于工作状态。

服务实例模式/VM 包含诸多优点。例如，每个微服务实例以完全隔离的方式运行，包含固定的 CPU 以及内存使用，且无须与其他微服务进行资源竞争。除此之外，该模式还可视为一种微服务开发过程中的技术封装。

当然，该模式也包含了自身的缺点，例如资源的利用率较低，每个服务实例都包含了整体 VM 开销，包括操作系统。该方案的另一个缺点是，由于使用 VM 引导操作系统的成本可能很高，因而部署和引导新版本的微服务通常较为缓慢。

图 13.18 显示了该策略的示意图。

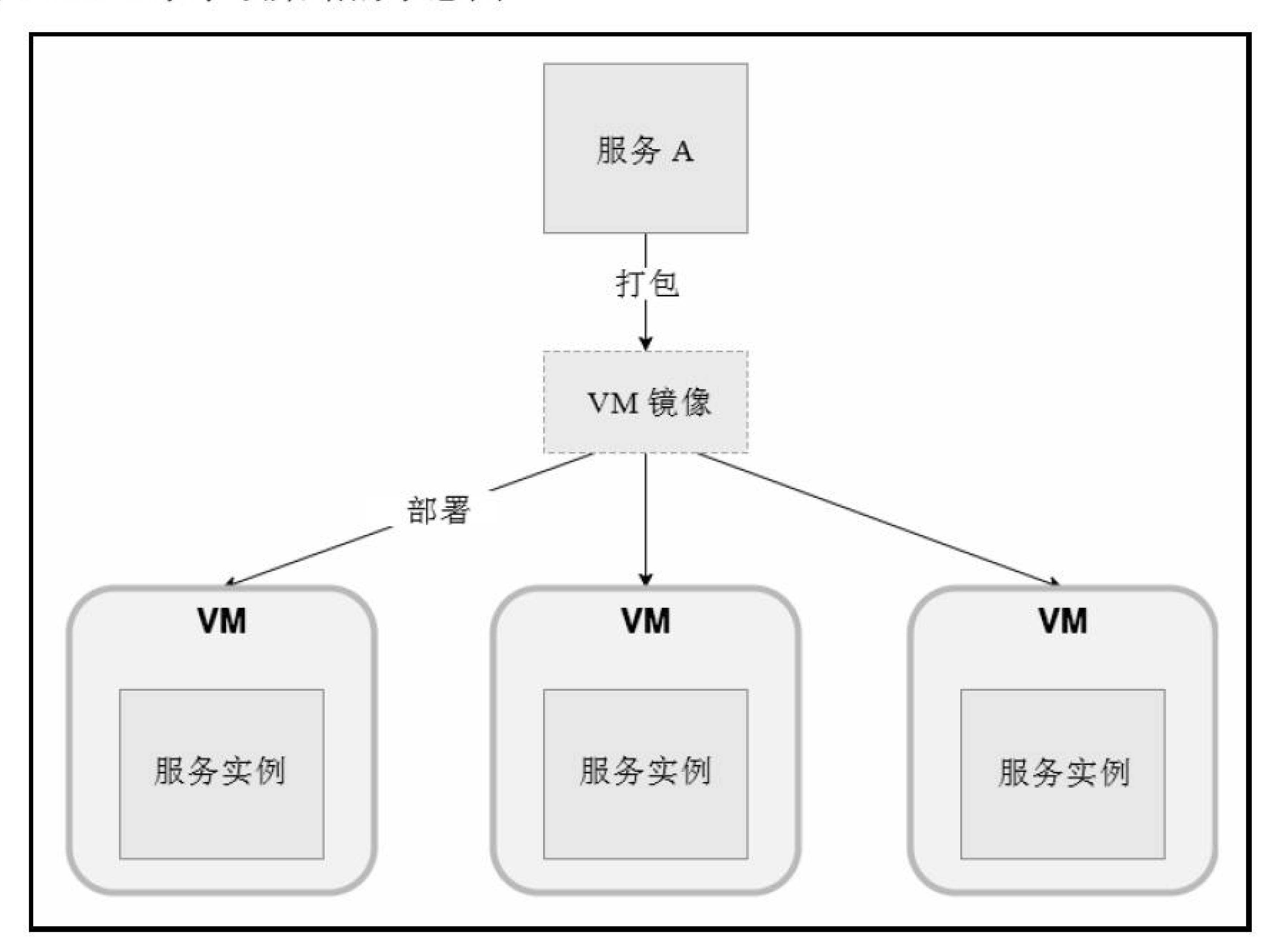


图 13.18

2．每个容器的服务实例

当使用服务实例/容器模式时，每个微服务实例都运行在实例特有的容器中。这里，容器表示为系统级运行的虚拟化引擎，如 Docker，我们已在本地开发项目中对其加以使用。

对此，我们需要使用到微服务的容器映像。在容器创建完毕后，即可发布一个或多个容器。通常情况下，多个容器将运行于每台物理或虚拟主机上。此处，建议使用集群管理器（如 Kubernetes）来管理容器。

服务实例/容器模式同样涵盖了自身的优缺点。容器的优点类似于虚拟机，并可隔离微服务实例。同时，容器的监视过程也较为简单。除此之外，容器也封装了微服务实现

过程中的相关技术方案，这一点与 VM 较为类似。与虚拟机不同，容器具有轻量级特征；容器映像的构建和初始化过程通常较快。

然而，容器的使用过程中也包含某些缺点。考虑到共享主机操作系统内核，因而其安全性与 VM 相比略逊一筹。除此之外，如果未采用提供相关机制操控容器的云平台，那么，容器基础设施建设的复杂度较高。

图 13.19 显示了在当前模式下容器的应用方式。

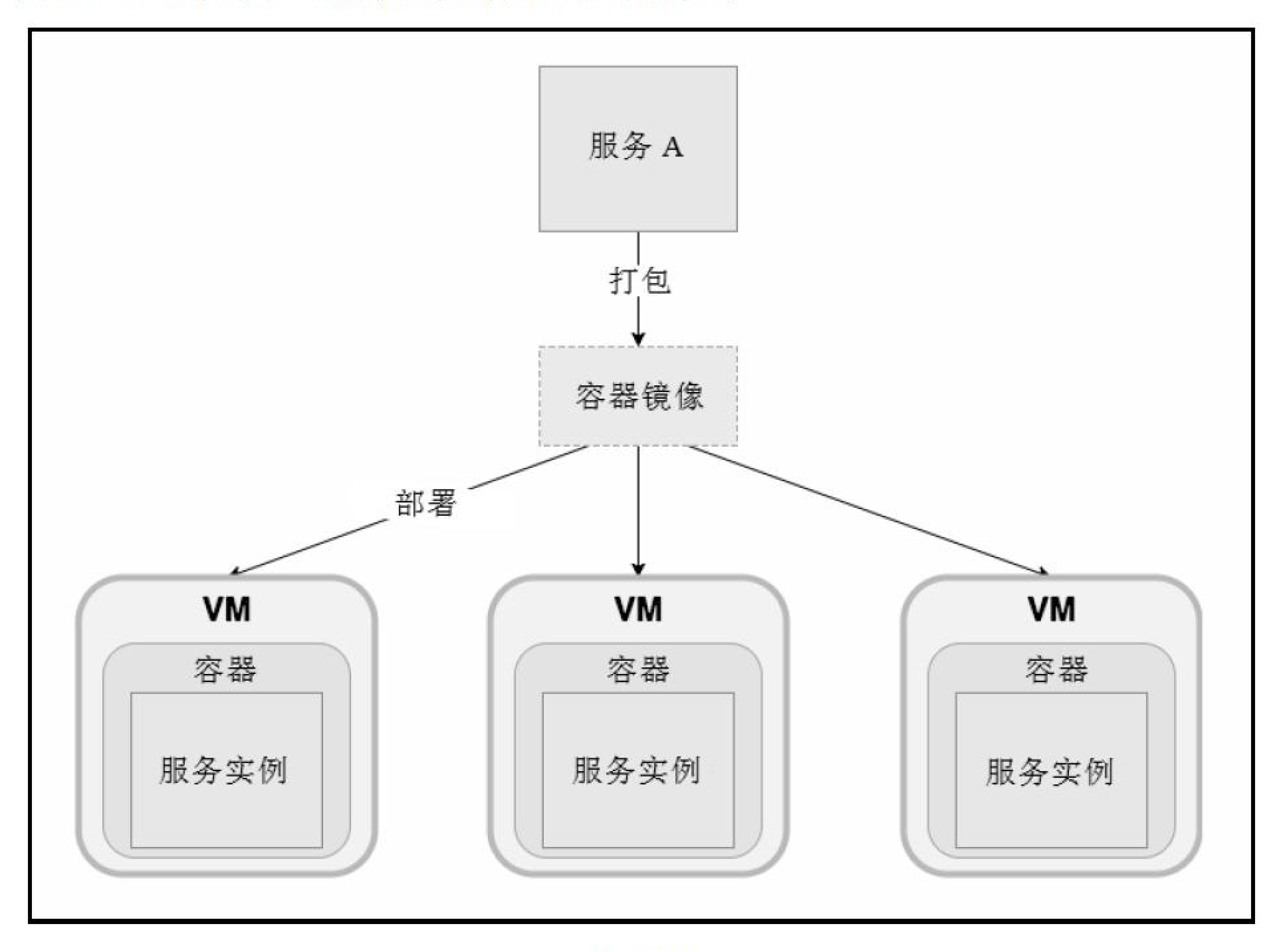


图 13.19

13.4　本 章 小 结

在前 12 章中，我们讨论了多种不同的模式以及最佳实践方案，甚至还涵盖了反模式方面的内容，其中涉及项目中的技术定义，产品中应用程序的部署和维护。

同时，我们也希望本书给读者带来愉快的阅读体验，以进一步扩展读者的知识层面。微服务架构的概念和模式仍处于不断发展中，对此，作者的建议是：继续保持学习状态。